



Modules for Hierarchical and Crosscutting Models

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von
Diplom-Informatiker

Klaus Ostermann
aus Cloppenburg

Referent: Prof. Dr. Mira Mezini
Korreferent: Prof. Gregor Kiczales

Tag der Einreichung: 14. April 2003
Tag der mündlichen Prüfung: 09. Juli 2003

Darmstadt
D17

Erklärung: Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Darmstadt, den 23. Juni 2003

*The three dots '...' here suppress
a lot of detail – maybe I should
have used four dots.*

Donald E. Knuth

Abstract

Good separation of concerns in software is the key for managing growing complexity. The most important task of programming languages with respect to this goal is to provide means to express the mental models of the domain experts as directly as possible in the programming language.

Since the advent of ‘structured programming’, programming languages feature modules for *hierarchical* models: We can view a software system at different levels of abstraction, based on whether we look at the interface of a module or zoom into the implementation of the module and the interfaces/implementations of the next lower-level modules. The possibility to view and implement a software system on different levels of detail has greatly improved the quality of software.

An important insight of recent years was the identification of so-called *crosscutting* concerns - concerns which cannot be localized in a given module structure or cannot be localized simultaneously with other concerns. Programming languages with explicit support for crosscutting concerns have already been proposed and implemented but this branch of language design is at the very beginning.

This is the context within which this thesis tries to improve the state-of-the-art. Based on the most successful paradigm for separation of concerns, object-oriented programming, a trio of proposals is described that refines and generalizes the conventional constructs for modularization and composition with respect to both hierarchical and crosscutting concerns.

The first two proposals deal with modules for hierarchical models. Firstly, the thesis goes back to the foundations of object-oriented programming and reasons about the relation between the two fundamental OO concepts for hierarchical decomposition: Inheritance and aggregation. There is a well-known tension between these two concepts: Inheritance enables incremental specification whereas aggregation allows polymorphic composition at runtime. Frequently, however, one needs a mixture of properties from both

aggregation and inheritance. *Compound references* are proposed, a new abstraction for object references that unifies aggregation, inheritance and delegation and provides explicit linguistic means for expressing and combining individual composition properties on-demand.

The second approach explores how the means for hierarchical decomposition can be generalized to work on sets of collaborating classes, motivated by the observation that a slice of behaviour affecting a set of collaborating classes is a better unit of organization and reuse than single classes. Different techniques and language extensions have been suggested to express such slices in programming languages but none of them fully fits in the conceptual framework of object technology. *Delegation layers* are a new approach to cope with these problems. It scales the object-oriented mechanisms for single objects, such as delegation, late binding, and subtype polymorphism, to sets of collaborating objects.

The third approach of the thesis is devoted to language concepts for representing *crosscutting models*, meaning independent models that represent different overlapping views of a common system. A crosscutting concern is seen as a concern that belongs to a different crosscutting model. Keeping crosscutting models independent allows the programmer to reason about each concern in isolation. The *Caesar* model is proposed, where ideas for hierarchical decomposition from the first two parts of the thesis are used and extended in order to provide means for representing and composing independent crosscutting models. Caesar's strengths are in the reuse and componentization of aspects. The notion of aspectual polymorphism as a generalization of subtype polymorphism to crosscutting models is introduced and a novel concept for dynamic deployment of aspects is proposed.

Zusammenfassung

Eine gute Separierung der verschiedenen Aspekte eines Programms ist der Schlüssel, um mit wachsender Komplexität umzugehen. Die wichtigste Aufgabe, die Programmiersprachen in Bezug auf dieses Ziel erfüllen sollten, ist, Mittel zur Verfügung zu stellen, um die mentalen Modelle der Domainexperten so direkt wie möglich in der Programmiersprache auszudrücken zu können.

Seit dem Anbruch der ‘Strukturierten Programmierung’ bieten Programmiersprachen Modulkonstrukte für *hierarchische* Modelle: Wir können ein Softwaresystem auf unterschiedlichen Abstraktionsebenen betrachten, je nachdem, ob wir nur die Schnittstelle eines Moduls oder auch seine Implementation und die Schnittstellen/Implementierungen der nächsten tieferliegenden Module betrachten. Die Möglichkeit, ein Softwaresystem auf unterschiedlichen Detailebenen zu betrachten und zu implementieren, hat die Qualität von Software entscheidend verbessert.

Eine wichtige Erkenntnis der vergangenen Jahre war die Identifizierung von sogenannten *crosscutting concerns* oder *querschneidenden Aspekten* - Aspekte, die in einer gegebenen Modulstruktur oder simultan mit anderen Aspekten im Programmtext nicht lokalisiert werden können. Erste Programmiersprachen mit expliziter Unterstützung für ‘crosscutting concerns’ wurden bereits entwickelt, doch dieser Forschungszweig ist noch im Entstehen.

Dies ist der Kontext, in dem diese Arbeit versucht, den gegenwärtigen Stand der Technik zu verbessern. Basierend auf dem erfolgreichsten Paradigma für die Separierung von Aspekten, objekt-orientierter Programmierung, werden drei Ansätze beschrieben, die die konventionellen Konzepte für Modularisierung und Komposition in Bezug auf hierarchische und querschneidende Aspekte verbessern.

Die ersten beiden Ansätze behandeln Module für hierarchische Modelle. Zunächst geht die Arbeit zurück zu den Grundlagen der objekt-orientierten

Programmierung und untersucht die Beziehung zwischen den beiden fundamentalen objekt-orientierten Konzepten für hierarchische Zerlegung: Vererbung und Aggregation. Es gibt eine wohlbekannte Spannung zwischen diesen beiden Konzepten: Vererbung ermöglicht inkrementelle Spezifikation, während Aggregation polymorphe Komposition zur Laufzeit ermöglicht. Häufig jedoch benötigt man eine Mixtur der Eigenschaften von Aggregation und Vererbung. Ein Modell namens *Compound References* wird vorgeschlagen. Eine Compound Reference ist eine neue Abstraktion für Objektreferenzen, die Aggregation, Vererbung und Delegation vereinheitlicht und explizite Sprachunterstützung dafür bietet, deklarativ je nach Bedarf gewünschte Kompositionseigenschaften festzulegen.

Im zweiten Ansatz geht es darum, wie die Mittel für hierarchische Dekomposition verallgemeinert werden können, um auch auf Mengen zusammenarbeitender Klassen zu arbeiten. Dies wird durch die Beobachtung motiviert, daß ein Modul, das das Verhalten mehrerer zusammenarbeitender Abstraktionen definiert, eine besser wiederverwendbare Organisationseinheit ist als einzelne Klassen. Unterschiedliche Techniken und Spracherweiterungen wurden bereits vorgeschlagen, um solche Module in Programmiersprachen ausdrücken zu können, doch keiner dieser Vorschläge passt vollständig in den konzeptuellen Rahmen der objekt-orientierten Programmierung. *Delegation Layers* sind ein neuer Ansatz, um mit diesem Problem umzugehen. In diesem Ansatz werden die objekt-orientierten Mechanismen für einzelne Objekte, zum Beispiel Delegation, späte Bindung und Subtyp-Polymorphie, auf Mengen zusammenarbeitender Klassen verallgemeinert.

Der dritte Ansatz dieser Arbeit behandelt Sprachkonzepte zur Repräsentation von querschneidenden Modellen. Mit ‘querschneidenden Modellen’ sind unabhängige Modelle gemeint, die unterschiedliche, sich überschneidende Sichten auf ein gemeinsames System repräsentieren. Ein querschneidender Aspekt wird als Aspekt gesehen, der zu einem anderen querschneidenden Modell gehört. Wenn diese querschneidenden Modelle unabhängig voneinander gehalten werden, wird es möglich, jeden Aspekt getrennt von den anderen zu behandeln. Das *Caesar* Modell wird vorgeschlagen, in dem die in den vorherigen Teilen vorgeschlagenen Ideen für hierarchische Dekomposition benutzt und erweitert werden, um die Repräsentation und Kombination unabhängiger querschneidender Modelle zu ermöglichen. Die Stärken von Caesar liegen in der Wiederverwendbarkeit und Komponentisierung von Aspekten sowie in der Möglichkeit, Aspekte polymorph zu benutzen. Das Konzept der Aspekt-Polymorphie wird als Verallgemeinerung der Subtyp-Polymorphie auf querschneidende Modelle eingeführt, und ein neues Konzept für den dynamischen Einsatz von Aspekten wird vorgeschlagen.

Contents

Preface	8
1 Introduction	9
1.1 Benefits of Proper Separation of Concerns	10
1.1.1 Comprehensibility	10
1.1.2 Reusability	10
1.1.3 Scalability	11
1.1.4 Maintainability	12
1.2 Hierarchical Decomposition of Concerns	12
1.3 Crosscutting Decomposition of Concerns	16
1.4 The Silver Bullet?	24
1.5 Thesis Organization	25
2 Hierarchical Decomposition: Single Entities	27
2.1 Introduction	27
2.2 Motivation	29
2.2.1 Composition Scenario 1: The Account Example	30
2.2.2 Composition Scenario 2: The Stream Example	32
2.2.3 Composition Scenario 3: The TextJustifier Example	34
2.2.4 Problem Statement Summary	36
2.3 Compound References	38
2.3.1 Field Methods and Overriding	38
2.3.2 Field Redirection with Compound References	40
2.3.3 Field Acquisition	44
2.3.4 Subtyping	48

2.3.5	Field Navigation	49
2.4	Evaluation of the Model	50
2.5	Reconciling dynamic specialization and static typing	54
2.6	Abstract Classes and Method Header Specializations	60
2.7	Related Work	62
2.8	Chapter Summary	65
3	Hierarchical Decomposition: Collaborating Entities	67
3.1	Introduction	67
3.2	Collaboration Composition and Mixin Layers	70
3.3	Delegation	74
3.4	Virtual Classes	76
3.5	Delegation Layers	78
3.6	Hot State and On-the-fly Extensions	80
3.7	Related Work	86
3.8	Chapter Summary	88
4	Encoding Crosscutting Models	90
4.1	Introduction	90
4.2	Problem Statement	95
4.3	Core Concepts	101
4.3.1	Collaboration Interfaces, their Implementations and Bindings	101
4.3.2	Wrapper Recycling	106
4.3.3	Composing Bindings and Implementations	107
4.3.4	Virtual Types	108
4.3.5	Object Constructors	110
4.3.6	Most Specific Wrappers	111
4.3.7	Interim Evaluation of the Model	111
4.4	Dimensions of Reuse	112
4.4.1	Component Type Hierarchies	113
4.4.2	Implementation Hierarchies	113
4.4.3	Binding Hierarchies	114
4.4.4	Polymorphism	120
4.4.5	Section Summary	121
4.5	Future Work: Layered Bindings and Implementations	122
4.6	Related Work	124
4.7	Chapter Summary	128
5	Combining Crosscutting Models	130
5.1	Introduction	131
5.2	Problem statement	133
5.3	Deploying Aspects With Caesar	138
5.3.1	Pointcuts and Advices	140

CONTENTS

5.3.2	Static and Dynamic Deployment	142
5.3.3	Dynamic Deployment and Concurrency	147
5.4	Evaluation	147
5.5	Related Work	151
5.6	Chapter Summary	153
6	Conclusions	154
6.1	Transparent Redirection	154
6.2	Incremental Specification, Increment Combination, and Subsumption	155
6.3	More Powerful Interfaces	157
6.4	Runtime Composition and Static Typing	158
6.5	Summary and Future Work	159

During the first four years of my computer science study in Bonn I was convinced that a Ph.D. is a pretty useless waste of time. In the industry, nobody cares for a Ph.D. Under the light of all the buzzwords that were drummed into our heads by the media - *start-up*, *going public*, *new market*, *B2B* etc. - an academic career seemed to be absurd anyway, and being a poor Ph.D. student for another three years looked like a dumb idea.

Luckily, I recognized - before it was too late - that a Ph.D. study should be viewed from a very different perspective: As a unique opportunity to think about a problem *really deep*. Through my industrial experience I knew that this is a privilege that I would probably *never* have in a regular job. I am happy that I finally decided to ‘do it’.

Many people have contributed to this thesis. First of all, I want to thank my thesis supervisor, Mira Mezini, for her extraordinary support. She “showed me the ropes” in the art of paper writing and scientific work. Despite her tight schedule, she was always willing to help in all conceivable ways. This thesis would look completely different without her.

I am indebted to Frank Buschmann, Lutz Dominick, Roman Pichler, Stefan Schulze, Christa Schwanninger, Egon Wuchner, and the whole Siemens CT SE 2 team for their support. In addition, I would like to express my gratitude to Siemens AG for their financial backup.

Gregor Kiczales, my co-supervisor, contributed with valuable comments and agreed to take on a 8000km journey just for the thesis defence.

Michael Eichberg, Jürgen Hallpap, Raquel de Moura Gurgel Silva, Walter-Augusto Werner, and Andreas Wittmann have implemented various proposals described in this thesis and provided important feedback concerning the feasibility and usefulness of the respective ideas.

Finally, I would like to thank my partner Annekathrin for her unrestricted backup and for moving twice (including the search for a new job) with me during the work on this thesis.

CHAPTER 1

Introduction

New language constructs for better separation of concerns in the development of software systems are needed! This is the claim that is backed up and explained in this chapter. First of all: What is meant by ‘good separation of concerns’?

A concern is any coherent issue in the problem domain. A concern might be an arbitrarily complex concept like ‘security’ or ‘pay-roll system’ or primitive concepts like ‘wait for mouse event’. Separation of concerns in software is closely related to composition and decomposition mechanisms in programming languages. Software composition and the reverse notion of software decomposition are about the partitioning of a software system into smaller parts (decomposition) and the assembly of software systems in terms of these smaller parts (composition). This definition does not say anything about *how* to decompose a software system. For example, dividing a software into two parts, whereby one part contains all even code line numbers, and the other part contains the odd code line numbers, would be a feasible software decomposition mechanism. However, obviously this decomposition would not make sense. A reasonable *criteria* to be used in decomposing systems is needed.

This is the point where the notion of *separation of concerns*, usually attributed to Parnas [Par72a] and Dijkstra [Dij76], proves useful. The separation of concerns principle postulates that every part of a decomposed software system should be responsible for a well-defined task or concern of the system. It should have as little knowledge about the other parts (and other concerns) of the system as possible such that it is feasible to reason about every concern in isolation.

1.1 Benefits of Proper Separation of Concerns

Proper separation of concerns has a number of benefits on different qualities. These effects are divided into four categories: *comprehensibility*, *reusability*, *scalability*, and *maintainability*.

1.1.1 Comprehensibility

“Our heads are so small that we cannot deal with multiple aspects simultaneously without getting confused.”

Edsger W. Dijkstra in [Dij76]

Probably the most obvious benefit of separation of concerns is that it becomes easier to *understand* a piece of software. If every program part can be understood on its own, we do not need to know the structure of the whole system in order to understand a part of it – we can concentrate on the concern in question while ignoring the other concerns. In the knowledge representation community, this problem is known as *delocalization* or *delocalized plans*, meaning that pieces of code that are conceptually related are physically located in non-contiguous parts of a program [SL86]. Different tools and techniques have been developed to ease knowledge discovery with delocalized plans, for example, better documentation techniques [LLP⁺88] or tool support [Wel95].

1.1.2 Reusability

“No matter how good our intentions, the first time we try to reuse something we discover a facet of the new problem the old module just can’t manage. So we tune it.”

Jack Ganssle, “The Failure of Reuse”, embedded.com, 2001.

Software reuse means that one piece of software is used in multiple places. The relation to separation of concerns is as follows: The more a piece of software concentrates on a single or few concerns, the more likely this piece of software can be reused in different contexts. Why is this so? A piece of software is all the more reusable if it contains as little dependencies on the context of usage as possible, and this is exactly the case if the software concentrates on a single concern. If the software in question encodes the concerns A, B, and C and we are only interested in concern B, then this piece of software is not reusable for our purpose.

Ubiquitous reuse leads to the software engineers’ dream of software that is assembled from a number of COTS (commercial-off-the-shelf) components, an idea that dates back to the late 1960’s [McI68]. The tremendous

effort that has been spent in commercial component models like Enterprise JavaBeans (EJB) or Corba Component Model (CCM) emphasizes the commercial importance of reusable software, see also [Szy98]. However, without means for good separation of concerns the dream of reusable software components is doomed.

1.1.3 Scalability

“Adding manpower to a late software project makes it later ”

Brooks’s Law

In this context, scalability refers to a reasonable relation between cost and size of a software system, i. e., the costs won’t explode in a big project. Separation of concerns fosters scalability in different ways:

- **Division of labor and knowledge:** If we are able to reason about every concern in isolation, we do not need omniscient geniuses but can instead assign experts in one particular area to the corresponding piece of software. This means that we can divide the labor into smaller pieces that are concurrently worked out by many different people.

The famous “Brooks’s Law” [Bro75] states that adding more people to a software project makes it later. In other words, the rule of three, $Time \approx \frac{Size}{\# People}$, does not apply to software engineering. The main reason for this observation is that the communication and coordination overhead increases quadratically with respect to the number of people involved. In a world with good separation of concerns we should be able to escape Brooks’s Law because the required communication and coordination is minimized.

- **Modular Checking:** From a compiler perspective, program parts with little dependencies on other program parts make it easier to perform *modular checking*, that is, check and compile a single part in isolation. However, this property does not only affect compiler scalability (which is rather negligible nowadays) – if a program part can be understood in isolation by the compiler, then it can also be understood by the programmer. This is also related to the question whether we have a *closed-world assumption* or an *open-world assumption*¹. Informally, the world assumption relates to whether we need to have global knowledge of all parts of a system (closed world) in order to check/understand/compile/reason about parts of a system.

¹The term *closed-world assumption* originally stems from artificial intelligence and refers to the assumption that everything which cannot be proved true is assumed as false.

Many modern concepts like runtime code loading, movable code, distributed computation etc. do not make sense if we have a closed-world assumption. Therefore, separation of concerns also contributes to the realization of these new concepts.

1.1.4 Maintainability

“There are many properties of objects that are non-local, for example, any kind of global consistency. What tends to happen in OOP is that every object has to encode its view of the global consistency condition, and do its part to help maintain the right global properties.”

William Cook

It is a well-known fact that maintenance of a software project frequently accounts for the largest part of the budget. Maintenance usually means adding, removing or changing a particular concern, e. g., changing the implementation of a feature. If we want to change a feature, we have two problems: a) *localizing* the feature and b) *updating* the implementation.

If a concern is not cleanly separated from the other concerns but instead spread over a large part of the system, the localization of a concern can be a real nightmare because we have to find all code snippets that contribute to the particular concern. The whole process of *reverse engineering* becomes much more complex if we have a software that has no good separation of concerns.

Updating such a tangled concern does also lead to problems because it is very difficult to make consistent changes such that the *integrity* of the software is preserved. We have to make sure that all changes in all different places are consistent with each other, which is highly non-trivial.

Similar observations apply to adding and removing concerns. In the first case, we have to find all code locations where code has to be inserted or changed, in the latter case we have to find all code locations where code has to be removed.

1.2 Hierarchical Decomposition of Concerns

In the 1960's and early 70's the need for good organization of software became apparent, due to more and more failures of large software projects. The obvious solution seemed to be to transfer the *divide and conquer* principle to software engineering: A large problem should be solved by solving many small problems. Wirth called this kind of software engineering “*program development by stepwise refinement*” [Wir71], meaning that

“In each step, one or several instructions of a given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language [...] A guideline in the process of step-wise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible.”

Already in 1968, Dijkstra wrote his influential article about the benefits of a layered architecture [Dij68]. Some years later, the new insights were also reflected in programming languages, leading to the development of structured programming [DDH72] and modules [Par72b, Wir82, DoD83].

The basic principles of modules can still be found in today’s programming languages: A module is a piece of software that may depend on other, lower-level modules, has some internal secrets, and provides a well-defined interface to other, higher-level modules. If we draw a dependency graph of a software system consisting of modules, we end up with a directed, acyclic graph (DAG). A DAG induces a hierarchy of layers because for every DAG with nodes V and edges E there exists a (not necessarily unique) number k and function $\lambda : V \rightarrow \{1, \dots, k\}$ such that $(u, v) \in E \Rightarrow \lambda(u) < \lambda(v)$ [BM76]. In other words, even if not explicitly intended by the programmer, software written in a conventional module-based language is always implicitly organized hierarchically. Of course, careful developers take advantage of modules in order to create the most useful hierarchies. For example, the layer pattern [BMR⁺96] describes an architecture within which a system is divided into layers, such that layer n depends only on layer $n - 1$, thereby making it very easy to replace individual layers.

Modern languages such as Java [AG96] also allow circular dependencies between modules, but the circles in the corresponding graph are usually very small: Sometimes circular dependencies make sense in order to encode ‘mind-sized’ components [CHP99], that is, components that are realized by a set of closely collaborating (hence mutually recursive) modules, but most analysis and design methodologies discourage the use of mutually recursive dependencies, see for instance Robert Martin’s *Acyclic Dependencies Principle* [Mar96]. Hence, a circle can be viewed as a single module and, from a high-level perspective, the claim is still true: today’s programming languages foster a hierarchical structure.

Physical hierarchical decomposition, that is, the physical assignment of source code to files, does not imply any particular semantic relation *per se*, for example conceptual specialization or containment, between the modules. However, from the perspective of clean separation of concerns, the implicit assumption of hierarchical decomposition is that it is possible to organize

the concerns of the system in a semantic hierarchy that can be more or less directly encoded in a corresponding module hierarchy. In terms of Wegner [Weg90], we want a direct correspondence between the *logical hierarchy* and the *physical hierarchy*. Winkler talks about the *concept-oriented view* versus the *program-oriented view* [Win92], Meyer postulates the *direct mapping principle* [Mey97], Larman calls for a *low representational gap* [Lar01] – many different names for the same idea.

If the conceptual and physical hierarchies of a system do not match, however, one or more modules of the system are *tangled* [KLM⁺97], meaning that they are responsible for more than one concern, or vice-versa, some concerns of the system are distributed over multiple modules. In other words, the separation of concerns principle is violated. We will have more to say about tangling in Sec. 1.3.

Whether or not it is possible to find a reasonable concern-to-module mapping depends greatly on the programming language used. For example, object-oriented programming languages have inheritance, which enables to represent semantic ‘is-a’ more easily and hence make it superior to other languages that do not have this feature.

However, before reasoning about better mappings from concepts to modules it makes sense to think about a reasonable modeling of concerns themselves. This is also in line with one of the favorite messages of the Turing Award winner and OO pioneer Kristen Nygaard, also emphasized on his last banquet speech at ECOOP 2002 in Malaga, shortly before he passed away: *To program is to understand and describe*.

If one considers this definition, one can draw conclusions from philosophical domains like ontology² and epistemology³ and scientific branches such as knowledge representation or cognitive science, and it quickly turns out that hierarchical modeling⁴ of concerns is a most natural process.

For example, in Aristotelian logic, the essential character of a subject is described by means of *genus*, a higher group to which the individual things belongs, and *difference*, i.e., what makes the subject different from other subjects with the same genus [Ros28]. In other words, Aristotle proposes to organize things in conceptual hierarchies. To *classify* things has been the most widely used approach to separation of concerns in the last 2000 years, leading to such diverse and powerful classification systems such as the biological classification system by Linnaeus or the International Classification of Diseases (ICD).

Nowadays, a wide range of relations that may hold between different subjects have been identified: Different flavors of conceptual specialization (for example, Chaffin identified four different flavors of conceptual special-

²The study of being or existence as well as the basic categories thereof. [Wik]

³The study of the origin, nature, and limits of human knowledge. [Wik]

⁴Please note that the term *hierarchical* refers to the DAG shape of the corresponding semantic net and not to the semantic relation ‘specialization’

ization [CH88], Brachman six [Bra83]), different flavors of ‘part-of’ relations (e.g., seven different flavors such as *component-of*(*Engine*, *Car*) and *portion-mass*(*Slice*, *Pie*) identified in [Sto93]), antonyms, instance-of relations etc.

Object-orientation is the programming paradigm whose decomposition mechanisms are most similar to those mentioned above: inheritance, aggregation, and instantiation. The first part of this thesis deals with the question of how well different flavors of conceptual specialization and ‘part-of’ relations can be modeled with inheritance and aggregation only. The result is a model called *compound references*, within which the strict separation of inheritance and aggregation is replaced by a smooth path from aggregation to inheritance, that is, a multitude of different semantic flavors in the interval between (possibly dynamic) inheritance and aggregation can be expressed, thereby enabling a better mapping of different specialization and part-of flavors to code.

The second part of the thesis deals with the problem that conventional hierarchical decomposition like in object-oriented languages is primarily about the description and relation of single entities. However, what happens if we try to include something that involves multiple entities? For example, the simple sentence

“Cato destroys Carthago”

involves at least three different things:

1. The person Cato
2. The city of Carthago
3. The destruction

Putting this behavior in any of the single entities abstractions would be arbitrary. One of the problems with conventional (object-oriented) hierarchical models is that they do not provide good means to model collaborative behavior. This is also reflected in the object-oriented view of a class as the implementation of an abstract data type: A set of data values and operations that operate on the data. This works quite well if one considers examples like stacks because all operations on a stack operate only on the stack and not on other data types. However, this does not work out if the operations that operate on the data involve multiple data types⁵.

The purpose of the second part of this thesis is to be able to apply the techniques and concepts to model single entities also to collaborative behavior. For example, it is desirable to be able to specialize a set of collaborating abstractions simultaneously in one module. In the *delegation layers* approach, collaborative behavior becomes a first class citizen in the

⁵This is also in line with a joke by Dijkstra who is said to have once stated that “abstract data types are a remarkable theory, whose purpose is to describe stacks.”

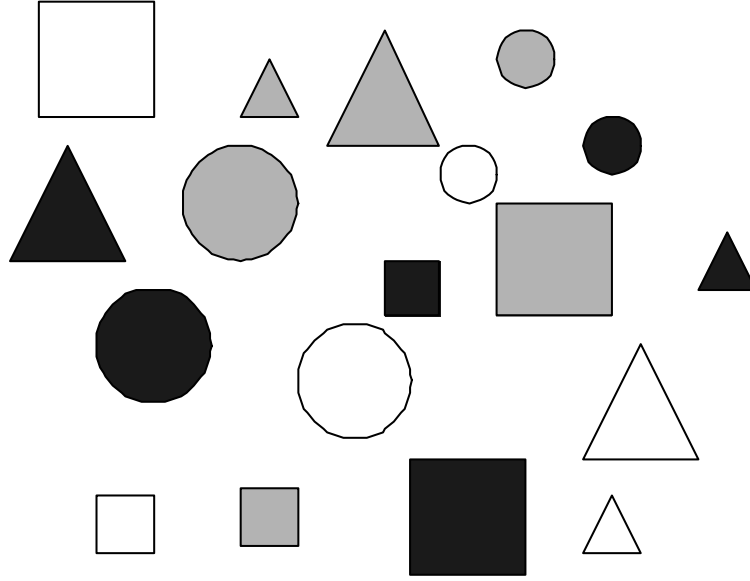


Figure 1.1: Abstract concern space

language, whereby the notions that proved so useful for single entities, such as subsumption and polymorphism, also apply to collaborative behavior.

1.3 Crosscutting Decomposition of Concerns

“Concepts in the real world, which programs and databases attempt to model, do not come in neatly packaged hierarchies.”

K. Baclawski and B. Indurkha in [BI94]

In this section, it is argued that there are certain modularity problems that will never be solved satisfactorily with hierarchical decomposition mechanisms only, how sophisticated they may ever be. This claim is also backed up by several other authors ([HO93, KLM⁺97, TOHS99, Ber90, AB92, ML98, BI94], to cite just a few).

In the author’s opinion, the main failure of hierarchical decomposition is that it assumes that real-world concepts have intuitive, mind-independent, preexisting concept hierarchies. However, our perception of the world depends heavily on the context from which it is viewed: There is no conceptual *lingua franca*. Lakoff [Lak90] argues that there is no objective model of reality:

“Knowledge, like truth, is relative to understanding. Our folk view of knowledge as being absolute comes from the same source

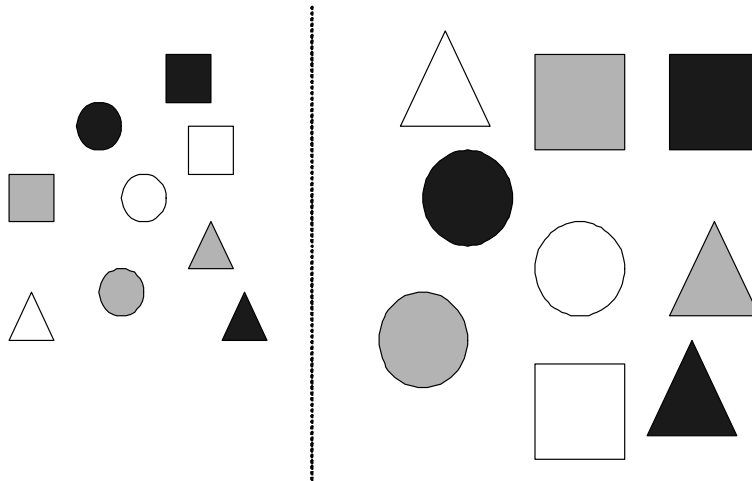


Figure 1.2: Divide by size

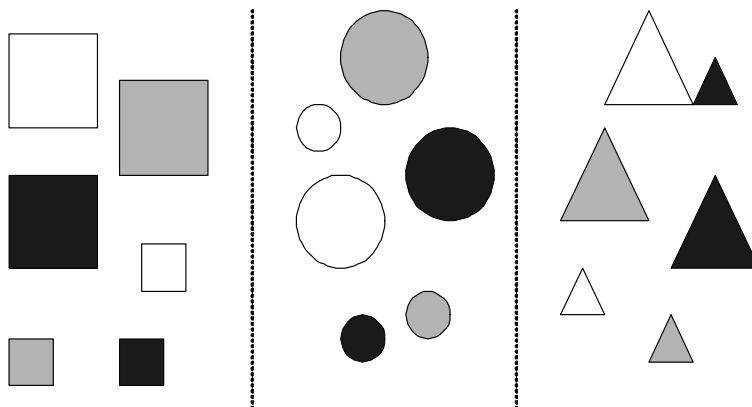


Figure 1.3: Divide by shape

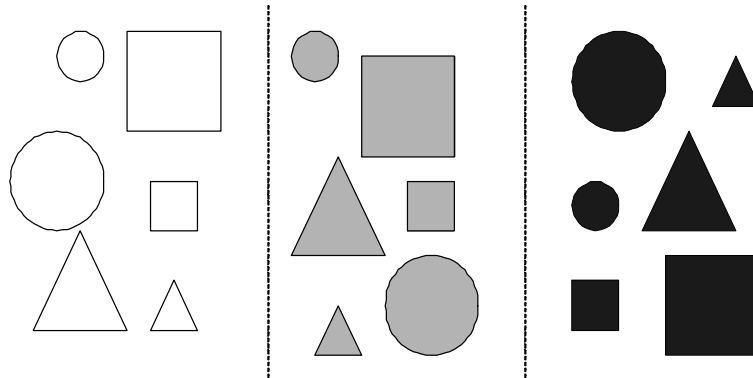


Figure 1.4: Divide by color

as our folk view that truth is absolute, which is the folk theory that there is only one way to understand a situation. When that folk theory fails, and we have multiple ways of understanding, or 'framing,' a situation, then knowledge, like truth, becomes relative to that understanding"

These observations also apply to software engineering. Every software system can be viewed from multiple different perspectives, and each of these perspectives may imply a different model of the concerns. The model which we choose to decompose our software system into modules has a big influence on the software engineering properties of the software. For example, Parnas observed that a data-centric decomposition eases changes in the representation of data structures [Par72a]. Tool abstraction [GKN92] makes it easy to modify functions of the system.

The problem can also be represented graphically. Each shape in Fig. 1.1 represents a particular concern of a software system. If we want to organize these concerns we could either divide them by size (Fig. 1.2), by shape (Fig. 1.3) or by color (Fig. 1.4). Each of these decompositions is equally reasonable. However, with a hierarchical modeling approach, we have to choose one fixed classification hierarchy – if we want to use multiple classification criteria, the criteria have to be organized in a sequence as in Fig. 1.5. In this example, the classification sequence was (*color, shape, size*). However, the problem is that with a classification sequence, only the first element of the list is localized whereas all other concerns are tangled in the resulting hierarchical structure. Fig. 1.5 illustrates this with the concern 'circle', which is crosscutting in the hierarchical decomposition. Only the color concern is cleanly separated into white, grey and black, but even this decomposition is not satisfactory because the color concern is still blended with other concerns. In the following, this problem will be referred to as the *arbitrariness*

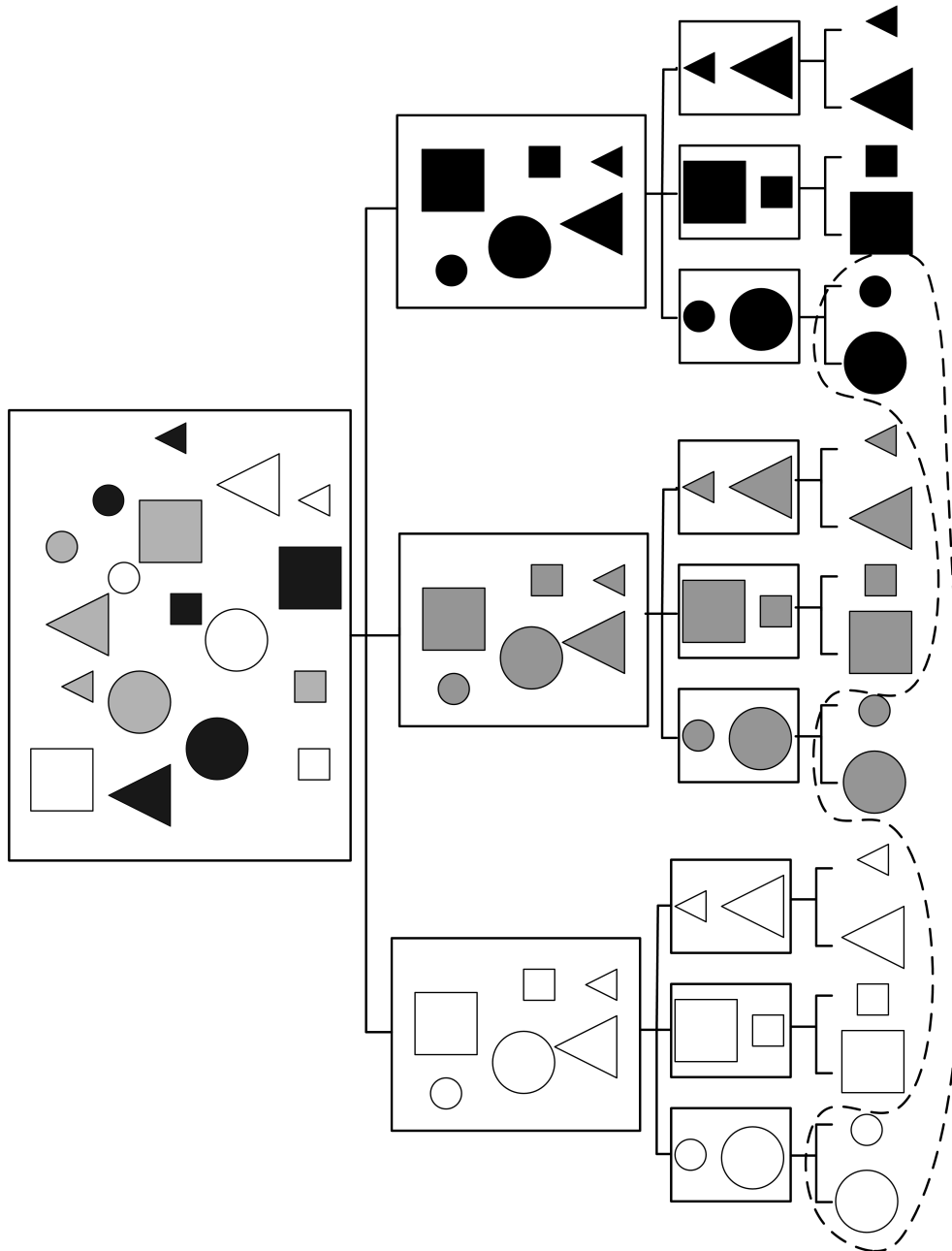


Figure 1.5: Arbitrariness of the primary model

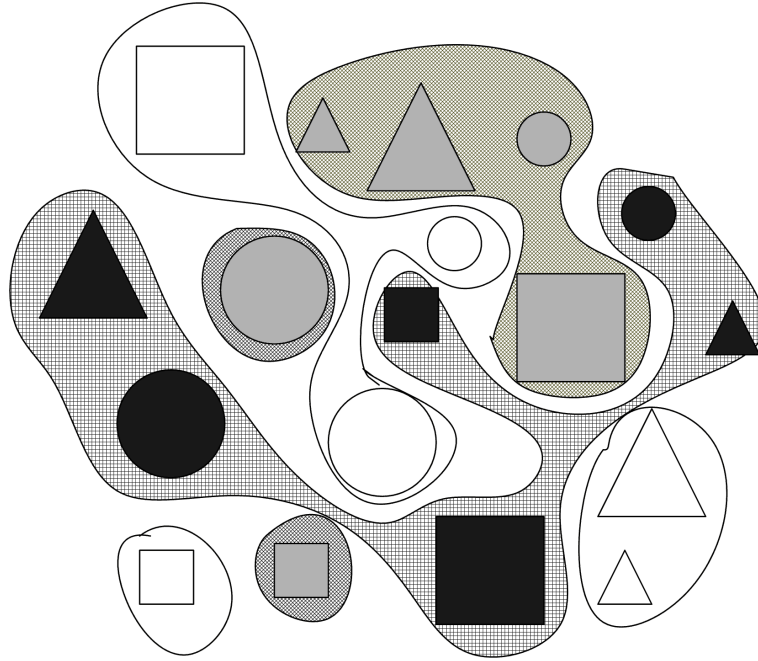


Figure 1.6: Projection of the ‘color’ decomposition

of the primary model.

The ‘arbitrariness of the primary model’ problem is related to the ‘tyranny of the dominant decomposition’ [TOHS99], meaning that “existing languages and formalisms generally provide only one, dominant dimension along which to separate - e.g., by object or by function” [TOHS99]. We prefer the term ‘arbitrariness of the primary model’ and its formulation as above, because it emphasizes that the problem arises not because only one dominant decomposition *mechanism*, such as classes versus functions, is supported. One can actually simulate functional decomposition in an object-oriented language [Wol97]. The problem is the primary mental model: Hierarchical decomposition requires a primary model and other models are subordinate as illustrated in Fig. 1.5.

With the conceptual framework introduced so far, *crosscutting* can be seen as a relation between two models. For a technical definition of the term “crosscutting”, the reader is referred to [MK03]; here we will give an informal (graphical) characterization of it.

We define crosscutting via *projections* of models. Fig. 1.6 shows a projection of the ‘color’ model from Fig. 1.4 onto the abstract concern space of Fig. 1.1. A projection of a model H is a partition of the concern space into subsets h_1, \dots, h_n such that each subset corresponds to an element of the model.

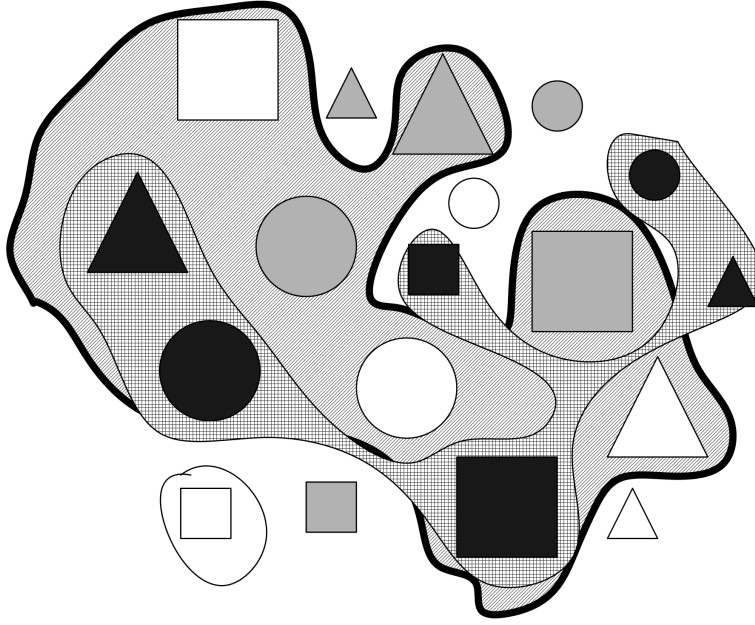


Figure 1.7: Crosscutting Hierarchies

Now, two models M and M' are said to be crosscutting, if there exist at least two sets o and o' from their respective projections, such that $o \cap o' \neq \emptyset$, and neither $o \subseteq o'$, nor $o' \subseteq o$. Fig. 1.7 illustrates how the **black** concern of the **color** model (Fig. 1.4) crosscuts the **big** concern of the **size** model (Fig. 1.2). These two concerns have in common the big, black shapes, but neither is a subset of the other: the black module contains also small black shapes, while the size model contains also non-black big shapes.

On the contrary, a model M is a *hierarchical refinement* of a model M' if their projections o_1, \dots, o_n and o'_1, \dots, o'_m are in a subset relation to each other as follows: there is a mapping $p : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ such that $\forall i \in \{1, \dots, n\} : o_i \subseteq o'_{p(i)}$. Crosscutting models are themselves not the problem, since they are inherent in the domains we model. The problem is that our languages and decomposition techniques do not (properly) support crosscutting modularity (see the discussion on decomposition arbitrariness above): If we have two crosscutting models M_1 and M_2 and make the decision to use M_1 as the primary model for our module structure, then the elements of M_2 cannot be cleanly localized in that module structure.

For a more concrete example, take a look at Fig. 1.8. It shows three different hierarchical decompositions of software for order processing. Each decomposition represents a different view on the system, a business logic decomposition on top, a decomposition from the viewpoint of security and finally at the bottom a persistence decomposition. The point is that with

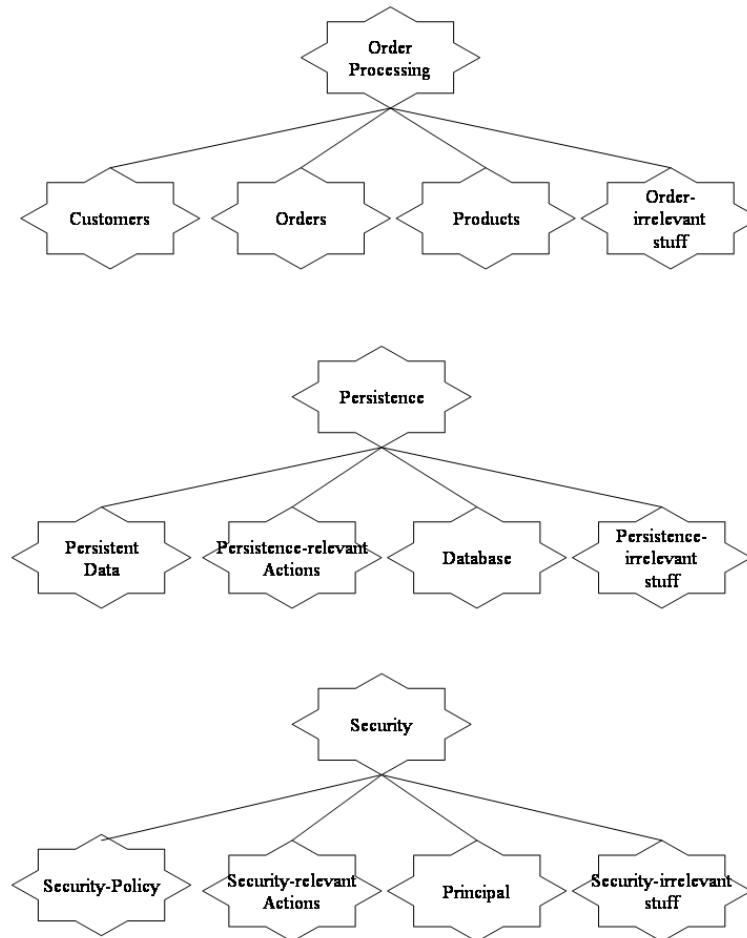


Figure 1.8: Different views of an order system

pure hierarchical decomposition mechanisms, a software developer has to choose *one* of these decompositions in order to organize his software correspondingly.

Most software developers would probably choose the business logic decomposition in order to organize the software modules because for most people it is the most ‘natural’ decomposition. However, the point is that each of these decompositions is *equally relevant*. Baclawski and Indurkha state: “We would like to affirm a stronger point of view in which there are no ‘standard’ conceptual hierarchies” [BI94].

Multiple inheritance has often been proposed and (mis-)used to model multiple decomposition criteria. For example, Meyer proposed an inheritance technique called *view inheritance* [Mey97, Sec. 24.10]. However, a model consists of multiple collaborating abstractions, whereas with multi-

ple inheritance we can only model a single abstraction. Hence it is not possible to encode crosscutting models appropriately with multiple inheritance. In addition, multiple inheritance has other well-known problems. First, multiple inheritance has some inherent conceptual difficulties (name collisions, diamond inheritance), for which no really satisfactory solutions exist (“Multiple Inheritance is good, but there is no good way to do it” [Syn87]). Second, the number of classes grows exponentially because one has to create a new class for every combination. Third, view inheritance is a maintenance nightmare, because the addition or removal of a classification criterion implies the invasive modification of all other code that refers to these classes. Last but not least, the integration of crosscutting hierarchies frequently implies sophisticated interactions between the different ‘views’ of an entity that cannot be appropriately expressed with multiple inheritance solutions like overriding, renaming etc.

As an extreme example, consider a simple concern such as logging of all I/O operations. With multiple inheritance, all classes that do I/O would have to be subclassed in order to override all methods that include I/O actions. Apart from the visibility problems (I/O might occur in private methods), the overhead of this approach would not be acceptable.

The third part of this thesis is concerned with decomposition of crosscutting concerns. A new language called *Caesar* is developed, within which it is possible to have multiple different decompositions *simultaneously* and to add new decompositions on-demand. Besides its features for crosscutting decomposition, a particularly interesting concept is its support for *aspectual polymorphism*, which can be seen as a generalization of usual object-oriented (subtype) polymorphism, by which polymorphism can be applied in *any* hierarchical decomposition, that is, polymorphism is available in every dimension of decomposition.

To make things more concrete, consider a hierarchy that divides figures into different shapes, as in Fig. 1.3⁶. This means that it is possible to write a method with a `Figure` parameter that is polymorphic with respect to the shape of the figure. Aspectual polymorphism means that polymorphism is available with respect to any hierarchical decomposition. In terms of the example this means that the code can simultaneously be used polymorphically with respect to color and size of figures as well! In terms of Fig. 1.8 this would imply that, for instance, we could choose and exchange the implementation of the persistence or security concerns at runtime.

Aspectual polymorphism is strongly related to the idea of *fluid aspect-oriented programming*, the “ability to temporarily shift a program (or other software model) to a different structure to do some piece of work with it, and then shift it back” [Kic01]. Let us see how usual object-oriented poly-

⁶In Fig. 1.3 the figures represented abstract concepts and not graphical figures but for the purpose of this example let us just consider them as graphical figures.

morphism via late binding fits into this picture: Calling a method $m(X\ x)$ with a parameter x that can be assigned polymorphically (i.e., x might refer to an instance of a subclass of X) at runtime can be seen as temporarily (during the execution of the method) shifting or transforming this method such that calls to methods of the parameter object are dispatched to the method implementations of the respective class of x . In this way, usual late binding can be seen as a primitive form of fluid AOP. It is limited because one can only transform with respect to our single decomposition hierarchy.

1.4 The Silver Bullet?

In 1987, Fred Brooks wrote his influential article “No Silver Bullet: Essence and Accidents of Software Engineering” [Bro87]. In this article, Brooks states:

“Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.”

He concludes that difficulties are inherent in the nature of software and thus software will always be hard: “There is inherently no silver bullet”.

To prevent a misunderstanding: This thesis is of course *not* Brooks’s silver bullet. However, it is worthwhile to reconsider the arguments that led to his pessimistic conclusion under the assumption that we have programming languages that enable better separation of concerns. He identifies four properties which he thinks are responsible for the inherent difficulty of building software:

- **Complexity.** Brooks states: “A scaling-up of a software entity is not merely a repetition of the same elements in larger sizes, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly.” The cause of this non-linear increase in complexity is due to a non-linear increase of dependencies, and dependencies explode due to messy separation of concerns! Hence, good separation of concerns can inherently lower the complexity.
- **Conformity.** In this case, Brooks identifies the conformance to multiple different interfaces as the problem: “Much of the complexity that [the software engineer] must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. These differ from interface to

interface, and from time to time, not because of necessity but only because they were designed by different people.” This problem is related to the arbitrariness of the primary model, identified in Sec. 1.3. The ability to adapt to a new decomposition (i.e., a new interface) easily, without modifying existing code, will substantially reduce the effort related to conformity.

- **Changeability.** The positive influence of separation of concerns on changeability and maintainability has already been discussed in Sec. 1.1.4.
- **Invisibility.** Brooks asserts that it is impossible to visualize software: “As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These graphs are usually not even planar, much less hierarchical.” This is basically just another description of the initial problem statement for motivating crosscutting decomposition: It should be possible to reason about any important viewpoint of the system in isolation, if this is true, it becomes also much easier to visualize the software.

This discussion should have made clear that better separation of concerns, both hierarchical and crosscutting, is really about reducing the *essential* difficulties of software engineering in Brooks’s sense.

1.5 Thesis Organization

Large parts of this thesis have previously been published and presented at conferences like ECOOP, OOPSLA, and AOSD. This fact had a big impact on the organization of the thesis.

Usually, a conference paper tackles a well-defined problem and presents a relatively self-contained solution. Recognizing that the number of people who are willing to read a Ph.D. thesis from front to cover is usually relatively small, the author tried to stick with the self-contained format of the conference papers in the sense that most chapters of this thesis can be read without too much dependence on previous chapters. This also means that this thesis does not present *the* big solution to one big problem but instead proposes a couple of small solutions for a number of small problems.

It should also be noted that this thesis does *not* propose an aggregate language that contains all language constructs proposed in all chapters of the thesis. Every language mechanism proposed here should be understood in the context of the problem it was introduced to solve.

However, this does not mean that the solutions proposed here are totally isolated from each other. In the technical chapters, there are many links to

other parts of the thesis that describe how one idea relates to another one or how one solution might benefit from another one. In addition, the last chapter of the thesis is devoted to the identification of recurrent themes or ‘red threads’ that pervade the thesis.

The remainder of this thesis is organized as follows: Chap. 2 and 3 are about better hierarchical decomposition mechanisms for single and collaborating entities, respectively. In Chap. 2 *compound references* are presented, a model within which the strict separation of inheritance and aggregation is replaced by a smooth path from aggregation to dynamic, object-based inheritance. Chap. 3 presents *delegation layers*, a model that scales well-known object-oriented concepts for single classes and objects, namely inheritance, subtyping and polymorphism, to sets of collaborating classes and objects.

Chap. 4 and 5 propose new modules for crosscutting models. Chap. 4 concentrates on the problem how crosscutting models can be represented and encoded, whereas Chap. 5 presents new ideas for the combination and deployment of crosscutting models with respect to aspectual polymorphism.

Chap. 6 concludes the thesis by a more abstract reflection on the different technical ideas underlying the models proposed. In particular, a number of recurring themes are identified that might be the topic of future research in the area.

CHAPTER 2

Hierarchical Decomposition: Single Entities

This chapter shares material with the paper ‘Object-Oriented Composition Untangled’ [OM01] which has been presented at OOPSLA 2001.

In this part of the thesis we try to release the tension between the conventional object-oriented mechanisms for hierarchical decomposition, inheritance and aggregation, by identifying and separating a set of properties that together characterize the difference between inheritance (both class-based and object-based) and aggregation. By providing dedicated language means for each of these properties, it is possible to build a seamless spectrum of composition semantics in the interval between object composition and inheritance.

2.1 Introduction

The two basic composition mechanisms of object-oriented languages, inheritance and object composition, are very different concepts, each characterized by a different set of properties. The properties of inheritance have been discussed in several works, e.g., [Sak89, Tai96, Mez98]. Also, the relationship between inheritance and object composition is carefully studied, e.g., in [Hau93, HOT97]. The mixture of composition properties supported by each mechanism is fixed in the language implementation and individual properties do not exist as abstractions at the language level.

However, often non-standard composition semantics is needed, with a mixture of properties, which is not as such provided by any of the standard techniques. In the absence of linguistic means for expressing and combining individual composition properties on-demand, such non-standard semantics

are simulated by complicated architectures that are sensitive to requirement changes and cannot easily be adapted without invalidating existing clients.

Actually, the need to combine properties of inheritance and object composition has already been the driving force for two families of non-standard approaches to object-oriented composition.

On one side, *delegation* [Lie86] enriches object composition with inheritance properties. Please note that in contrast to the frequent use of the term *delegation* as a synonym for forwarding semantics, in this thesis it stands for dynamic, object-based inheritance. In pure delegation-based models, objects are created by cloning other prototype objects, and objects may inherit from other objects, called *parents*. Hence, in such models one has object composition and delegation, but no class-based inheritance. The most prominent programming language in this family is SELF [US87]. More recently delegation-based techniques are integrated into statically typed, class-based languages, which thus provide class-based inheritance, delegation, and object composition [Kni99, Ern99, BW00]. On the other side, several *mixin*-based models [BC90, Mez97, FKF98, ALZ00] approach the goal of combining inheritance and object composition properties from the opposite direction, enriching inheritance with object composition properties, such as the ability to statically/dynamically apply a subclass to several base classes.

Like standard composition mechanisms, these approaches also do not provide abstractions for explicitly expressing individual composition properties that would allow to combine these properties on-demand. In this chapter, we distinguish between five properties that can be used to describe the relation that holds between two modules M and B (classes and/or objects) to be composed, whereby B denotes the base module, M denotes the modification module, and $M(B)$ denotes the composition.

1. **Overriding:** The ability of the modification to override methods defined in the base. In $M(B)$, M 's definitions hide B 's definitions with the same name. Self-invocations within B ignore redefinitions in M .
2. **Transparent redirection:** The ability to transparently redirect B 's `this` to denote $M(B)$ within the composition.
3. **Acquisition:** The ability to use definitions in B as if these were local methods in $M(B)$ (transparent forwarding of services from M to B).
4. **Subtyping:** The promise that $M(B)$ fulfills the contract specified by B , or that $M(B)$ can be used everywhere B is expected.
5. **Polymorphism:** The ability to (dynamically or statically) apply M to any subtype of B .

	inheritance	object composition	delegation	mixin inheritance
overriding	x	-	x	x
redirection	x	-	x	x
acquisition	x	-	x	x
subtyping	x	-	x	x
polymorphic	-	dynamically	dynamically	statically

Table 2.1: Composition properties supported by standard mechanisms

Table 2.1 shows the set of properties which will be discussed as row indexes. Columns are indexed by existing object-oriented composition mechanisms.

The key idea of the approach presented in this chapter is the *separation* and *independent applicability* of these notions by providing explicit linguistic means to express them. This allows the programmer to build a seamless spectrum of composition semantics in the interval between object composition and inheritance, depending on the requirements at hand, making object-oriented programs more understandable, due to explicitly expressed design decisions, and less sensitive to requirement changes, due to the seamless transition from one composition semantics to another.

The remainder of the chapter is organized as follows. Sec. 2.2 discusses examples where non-standard combinations of composition properties are desirable. Sec. 2.3 presents the basic concepts of our model. The model is evaluated in Sec. 2.4. Sec. 2.5 discusses some advanced issues related to static type safety. Sec. 2.6 discusses some issues related to abstract classes and method header specialization. Related work is discussed in Sec. 2.7. Sec. 2.8 summarizes the chapter and suggests areas of future work.

2.2 Motivation

In this section we consider three composition scenarios where non-standard combinations of composition properties make sense.

In all cases, we discuss various designs that can be used to achieve the desired composition semantics. However, please note that this section is *not* about proposing *the* ultimate designs for the given scenarios. The reader might eventually come up with other, equivalent or even superior, designs to the same scenarios. Yet, this is not essential for the purpose of this chapter: The main message that should be conveyed is rather (a) that in all cases *some* sophisticated design is needed, which does not explicitly state important conceptual relationships between the involved abstractions, and (b)

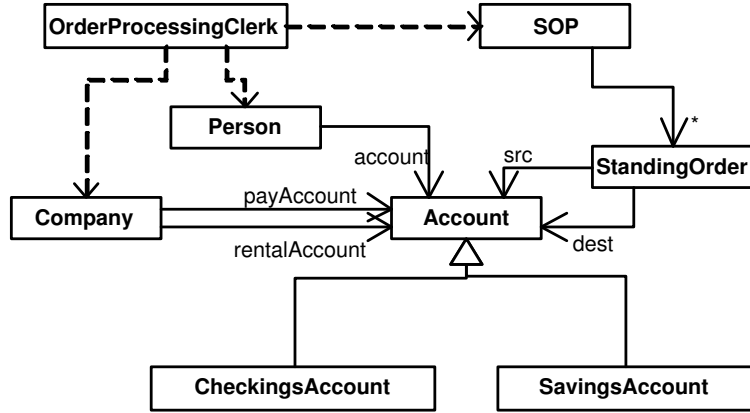


Figure 2.1: Class diagram for account example

that different designs are needed for different combinations of composition properties.

2.2.1 Composition Scenario 1: The Account Example

Consider an application in the banking domain with persons, companies, accounts, and standing orders. The relation between persons/companies and accounts is usually one to many. However, in this example each account should have a dedicated role for its owner. For example, a company should have a dedicated pay account and a dedicated rental account. This makes it possible to choose the appropriate account for specific transfers automatically. In this (simplified) example, a **Person** has only one “main” account and a **Company** has a rental and a pay account. Different kinds of accounts exist (**SavingsAccount**, **CheckingsAccount**), and accounts are subject to frequent changes at runtime. A particular account may be shared, as e.g., two persons may use the same account, or the pay account and the rental account of a company may be identical. A class **SOP** (standing order processing) is used for the registration, deregistration and execution of standing orders. On execution, multiple standing orders with identical source and target accounts are summarized to a single transfer.

A class diagram for this problem is shown in Fig. 2.1. Based on the information on a pay order, the **OrderProcessingClerk** gets the account objects from the involved **Person/Company**, creates a **StandingOrder** and registers it with a **SOP**. This design is simple and easy to understand. However, it has a problem: If the account of a person changes, a previously registered standing order will still be executed with respect to the outdated account. With the design in Fig. 2.1, one has to update all account references that

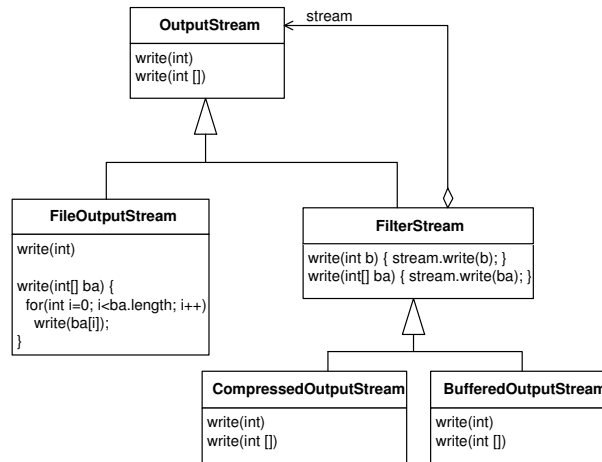


Figure 2.2: Decorator design for output streams

were ever given out by a person or company manually. That is definitely undesirable.

Given a reference `p` to a `Person` object, one ideally wants “`p`’s account”, i.e., a compound, or indirect reference to `account` via `p`, to be passed to `SOP`, rather than the `account` reference itself. In other words, some kind of *redirect semantics* for references is desirable: the meaning of `account` should be late bound within the current context of the object referred to by `p`, whenever “`p`’s account” gets evaluated. Due to the lack of such *compound references* in standard object-oriented languages, the architecture of the design has to be changed in order to simulate them. Some possible solutions are discussed below.

- A *decorator* [GHJV95] that contains an account object and forwards all calls to it is passed to `SOP` instead of the account object itself. The base object of the decorator (the account) can be changed without the need for further manual updates. However, the identity test in `SOP` fails: If two persons share an account, `SOP` compares non-identical decorators with the same base object. Other subtleties of an architecture that uses the decorator pattern for composition are highlighted in the next scenario.
- A second approach is to change the `SOP` class so that it accepts `AccountOwner` instead of `Account` objects with `AccountOwner` being an interface with a single `getAccount()` method. `Person` and `Company` have to implement the `AccountOwner` interface. This is difficult in the `Company` case, because a company has two different accounts. For this reason, one has to create a separate `AccountOwner` subclass for `Com-`

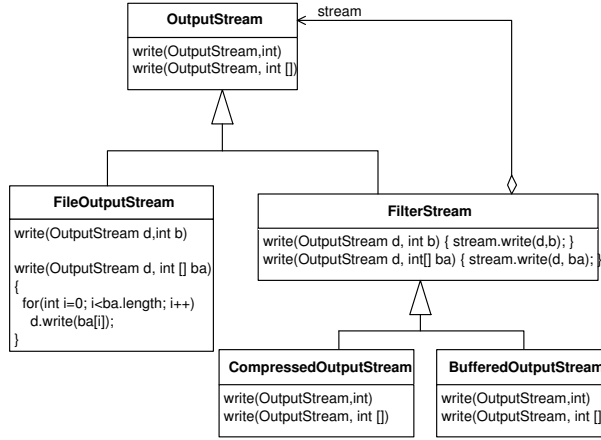


Figure 2.3: Simulation of transparent redirection

pany¹. Besides its complexity, the main drawback of this approach is that we have to modify the `StandingOrderProcessing` class, which might not be desirable or even possible in case this class is purchased as part of a banking component library. Another limitation of this proposal is that it works only for one level of redirection. For example, one might want to register a standing order that transfers money to the account of a person's current spouse (and, spouses are also subject to frequent changes these days).

- A common approach to avoid the coupling of the sender of a request to its receiver is the chain of responsibility pattern [GHJV95]. Applied to our example this would mean that each account has an optional successor account and new accounts are appended to the previous account. Calls to the account are forwarded to the last and current account in the chain. Besides its considerable complexity, this approach is not compatible with sharing of accounts.
- Another possible solution would be to let the `SOP` class be an observer [GHJV95] of persons and companies that is notified whenever an account is exchanged. However, it is easy to see that this would result in a design that is even more complicated than the previous ones.

2.2.2 Composition Scenario 2: The Stream Example

I/O Streams exist in multiple variations and different stream features are typically implemented as decorators [GHJV95] of a basic stream class (see

¹In Java, these classes would probably be implemented as inner classes.

e.g., the Java I/O package [Sun]), so that the set of desired features for a stream instance can be chosen dynamically. A typical decorator design for output streams is shown in Fig. 2.2.

The goal of using the decorator to compose basic `OutputStream` functionality with optional filtering features is to achieve the following composition properties: (a) subtyping between the resulting composition and the base component, (b) acquisition of the base behavior within the filtering functionality, (c) dynamic polymorphism - a certain filtering should be applicable to any subtype of `OutputStream`, and (d) overriding of the methods that have to be changed for the extended functionality. The decorator pattern realizes dynamic polymorphism of the composition by means of object composition and subtype polymorphism. Acquisition and overriding is achieved by implementing the base component's interface by means of forwarding methods resp. decorator-specific methods. The decorator becomes a subtype of the component by inheritance. The simulation of these composition properties, however, has a number of shortcomings:

- The implementation of the decorator class is a tedious and error-prone work, due to the manual simulation of the acquisition feature. In addition, it suffers from the *syntactic fragile base class problem* [Szy98]: Whenever the interface of the base component changes, the corresponding forwarding methods have to be added or deleted.
- There is no transparent redirection. This means that method calls to `this` within the base component are not dispatched to their overridden methods in the decorators but to the local implementations. A further consequence is that if a base object passes `this` to other objects, it passes itself instead of the decorator. In some situations, however, the opposite effect would be desirable. This anomaly is known as the *self problem* [Lie86] or as *broken delegation* [HOT97]. The manual simulation of transparent redirection is rather complex and leads to a design that is very different from the original decorator pattern because the base object needs some way of knowing about the decorators. One alternative is to store a back reference to the decorator in the base, but this would prohibit multiple decorators for one base object. Another solution with (again) a different design is to pass the decorator to the base object in every method call. A corresponding design is illustrated in Fig. 2.3.
- The base class may define state. All decorators inherit this state and become unnecessary heavy. Although merely a subtype relationship between the decorator and the base component is needed, the decorator is enforced to inherit the state, due to the use of inheritance for subtyping. This is usually no problem if the usage as a base for decorators was already anticipated at writing time. However, if a predefined

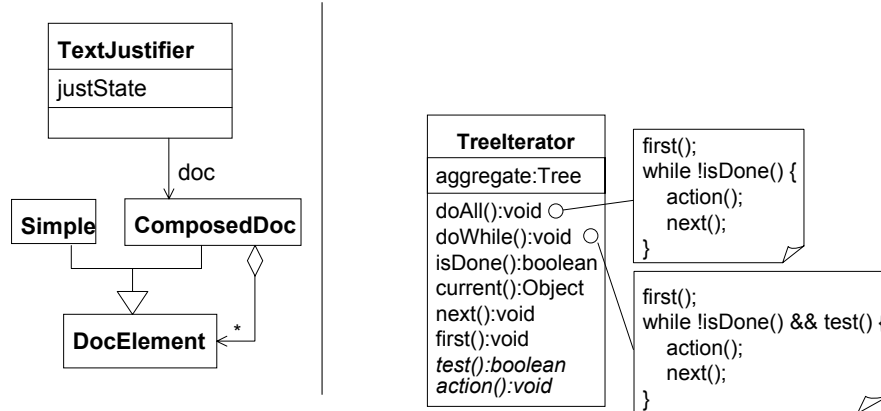


Figure 2.4: Structure of text justifier and tree iterator

library class should be decorated, this may be a problem.

2.2.3 Composition Scenario 3: The TextJustifier Example

Envisage a **TextJustifier** command class in a text processing system, which justifies all paragraphs in a document, except for preformatted paragraphs. The document elements to be justified are stored in a recursive object structure, as shown in the diagram on the left-hand side of Fig. 2.4². For performing the document justification the text justifier needs to iterate over the document structure. Assume that a tree iterator class as shown on the right-hand side of Fig. 2.4 has already been implemented. The class **TreeIterator** encodes a breadth-first iteration strategy for recursive object structures. It can be used by overriding the **action()** and **test()** methods for the specific purpose. The iterator class provides a number of iteration mechanisms, e.g., applying **action()** to all elements that satisfy **test()** (**doAll()**), or up to the first one that does not satisfy **test()** (**doWhile()**), and so on.

Assume that the design shown in Fig. 2.5³ where text justification and iteration functionality are composed by means of inheritance is just good enough for satisfying the requirements on our system during an early stage of the development process. In light-weight processes such as Extreme Programming [BFK00] where refactoring plays a key role the “make it work first and then make it better” philosophy encourages such an iterative de-

²In a more realistic situation, one would have to apply the visitor pattern to connect **TextJustifier** and the **DocElement** hierarchy. For the sake of simplicity, it is assumed that this is not the case in our example. The problems discussed here apply to a visitor-based design as well.

³In the design it is assumed that **DocElement** implements **Tree**

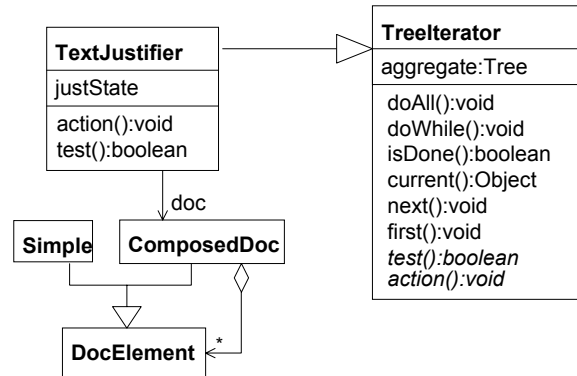


Figure 2.5: Usage of the iterator by inheritance

velopment.

In a later iteration stage it becomes apparent that inheritance is not the correct composition semantics anymore. First, **TextJustifier** should not be a subtype of **TreeIterator** anymore because a **TextJustifier** is not a special kind of an iterator. In addition, the acquisition semantics that comes with inheritance is not desired anymore; all methods of **TreeIterator** pollute the interface of **TextJustifier**, which has become complex anyway during the development. Second, the initial requirements have slightly changed: It should be possible to determine the iteration strategy to be used with a **TextJustifier** at runtime. For this purpose, subclasses **PreOrder** and **PostOrder** of **TreeIterator** have been implemented that refine the default breadth-first semantics by overriding the `first()` and `next()` methods.

Now, the question is how to compose the text justifier in Fig. 2.4 with the iteration hierarchy, such that the above set of composition properties are satisfied. A feasible solution is schematically presented in Fig. 2.6. **TextJustifier** has an instance variable, `it`, of type **TreeIterator**, which can be assigned to an instance of **MyPreOrder**, **MyIterator**, or **MyPostOrder**. The latter are defined as subclasses of the corresponding library classes and redundantly implement the `test()` and `action()` methods for the justification purposes. It is quite reasonable to assume that the test and the action performed in each step of the iteration needs information from the text justifier object, which is provided via the `context` reference on the **TextJustifier**.

Obviously, the design in Fig. 2.6 is very different from the predecessor design in Fig. 2.5. That is, two different mixtures of features for composing the same pieces of functionality are realized by two very different designs. Furthermore, the design is more complex than the design in Fig. 2.5, and

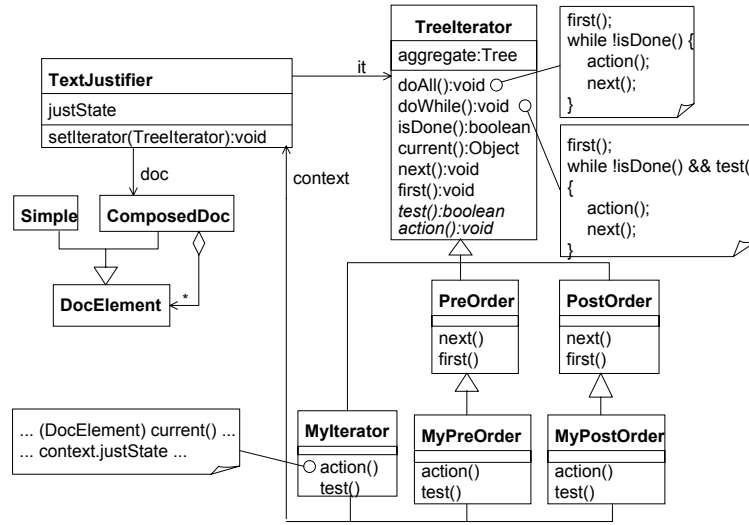


Figure 2.6: Initial design for dynamic composition

it does not reflect the conceptual relationships between the entities in it. Additional classes and associations have been introduced, and the **MyXXX** classes contain duplicated implementations of `action()` and `test()`.

At this point, it becomes clear that the initial iterator design is unsatisfactory because it leads to code duplication as in Fig. 2.6. It would have been better to choose a more sophisticated design for the iterator classes right from start, namely iterators that use a command class (*IterationStep*) as shown in Fig. 2.7. But this still does not solve the problem: A complex design such as in Fig. 2.7 is still required, although the conceptual relationship between **TextJustifier** and **TreeIterator** is as simple as the initial design in Fig 2.5. The design in Fig 2.5 would already be sufficient, even for the dynamic composition, if only it would be possible to configure the relation between **TextJustifier** and **TreeIterator** with the properties in Tab. 2.1.

2.2.4 Problem Statement Summary

So far so good. In all cases, the desired composition semantics can indeed be achieved somehow. Still, the result is highly unsatisfactory. Why? First, the most severe problem is that the architectures we ended up with are completely different, depending on the desired mixture of composition properties. Second, the design gets complex, as soon as a composition is required that deviates from the semantics of the standard composition mechanisms directly supported by linguistic means.

As illustrated by the first composition scenario, different programmers

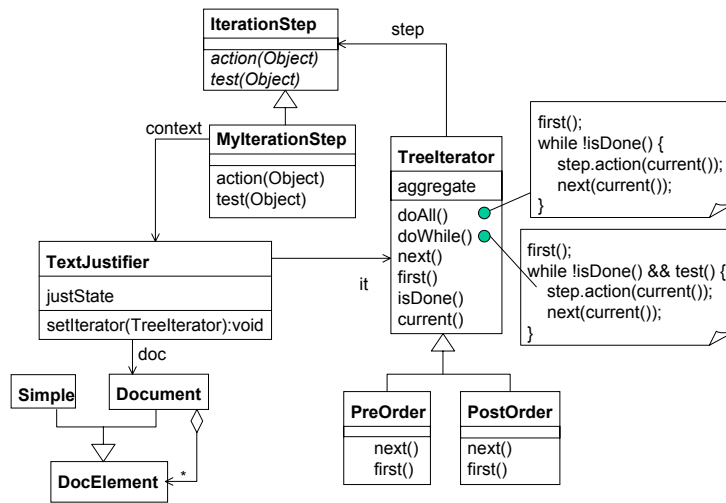


Figure 2.7: More sophisticated design for dynamic composition

may come up with different architectures even for the same composition semantics. Moreover, any later change of the composition features might require switching to another architecture. This is not only a tough challenge for any programmer. It also affects the understandability, and hence maintenance of object-oriented programs: The very important knowledge about the encoded composition semantics is not explicitly expressed by any of the designs that simulate non-standard semantics. In general, it is not obvious how to separate the part of the architecture that is directly involved in encoding application logic from the part of the architecture that serves as an infrastructure for encoding non-standard composition semantics. As a consequence, it is hard to guess from looking at the design that two architectures are different merely because they encode different composition semantics, or that two different architectures actually implement the same application logic, and only differ in the way they encode the same composition semantics.

The frequency of changes in the composition features is documented by *refactorings*, such as, “Replace Delegation with Inheritance”⁴, “Replace Inheritance with Delegation”, “Hide Delegate” and “Remove Middle Man” [Fow99]. In the terminology of this model, each of these refactorings can be seen as moving from an architecture with a certain mixture of composition properties to another one with another mixture of composition features that better fits the requirements or the current state of the development process. The work on refactoring recognizes that such transformations are not trivial and aims at aiding programmers in performing them by describing the

⁴Fowler uses the term delegation in the sense of decorator-like forwarding

process in a systematic way, or even (partly) automating it by means of refactoring browsers. The highly positive echo that the work on refactoring has found in the object-oriented community, especially in the practice of everyday programming, actually supports the claim that the need for different architectures to express different composition semantics makes a programmer's life harder.

However, this model follows another path in approaching the problem, motivated by the observation that identifying and describing common refactorings does not solve the core problem: It does not change anything in the fact that different architectures for different composition semantics are needed and one still needs to switch from one architecture to the other in order to react to requirement changes. Consequently, the emphasis is put on tackling the problem in its roots: At language design. This model is based on the claim that besides identifying and describing refactorings one should strive for language mechanisms that make some of the refactorings obsolete, or at least explicit in the language. This requirement becomes even more relevant in a component setting where refactoring steps like “adjust all clients to call the new server” are no longer feasible.

2.3 The Compound Reference Model

This section introduces the basic notions of our model as an extension of the Java programming language [AG96]. However, the concepts are easily applicable to other statically typed OO languages. Each introduced feature corresponds to a row in Table 2.1 and represents a step forward on a seamless transition from object to inheritance-based composition semantics.

2.3.1 Field Methods and Overriding

The operational semantics of the model is explained with the notion of *field methods*. A *field method* is a method that pertains to a specific field. Syntactically, the affiliation of a method to a field is expressed by prefixing the method name with the field name using “.” as separator. A class *C* with a field *f* of type *F* can be thought of as implicitly containing a method named *f.m()* for every public method *m()* of *F*. The method named *f.m()* has the same signature as *m()* in *F*, and its visibility is identical to the visibility of *f* in *C*. The default implementation of *f.m()* in *C* is one that simply forwards *m()* to the object referred to by *f*, denoted within the implementation of *f.m()* by the special pseudo-variable *field*. This is similar to the pseudo-variable *super* denoting an overridden method within the overriding method. Finally, any invocation of *m()* on the object referred to by *f* within *C* should be thought of as being dispatched to the corresponding implicit


```
class TextJustifier {
    private TreeIterator it;
    public void justify() { ... it.doAll() ... }

    // ** begin of implicitly available field methods **

    private void it.doAll()          { field.doAll();   }
    private void it.doWhile()        { field.doWhile(); }
    private void it.doUntil()        { field.doUntil(); }
    ...
    private void it.action(Item x)   { field.action(x); }
    private boolean it.test(Item x) {
        return field.test(x);
    }
    // ** end of implicitly available field methods **
}
```

Figure 2.8: Implicit field methods in TextJustifier

```
class TextJustifier {
    private TreeIterator it;

    private void it.doAll() { ... }

    public void justify() {... it.doAll(...); }
}
```

Figure 2.9: Explicit field methods in TextJustifier

field method `f.m()`.

For illustration, recall the iterator example from Sec. 2.2. With the implicit field methods written down, the code for the `TextJustifier` would look like in Fig. 2.8⁵. The call `it.doAll()` within `justify()` should be thought of as actually calling the implicitly available field method named `it.doAll()`.

Until now, the introduction of field methods into the implementation of a class has no impact on the semantics of the class. The `TextJustifier` implementation presented in Fig. 2.8 is semantically equivalent to an implementation that does not contain any implicit field method. The decisive point is that implicit methods can be replaced by explicitly available methods. For example, in order to implement an action to be undertaken whenever `it.doAll()` is called in `TextJustifier`, the programmer of `TextJustifier` would implement an explicit field method, called `it.doAll()`,

⁵In the context of this section, the reader should think of the abstract methods as being implemented empty.

encoding the desired behavior, as shown in Fig. 2.9. Please note that Fig. 2.9 is only an illustration of explicit field methods and not the final solution for the TextJustifier problem.

Before leaving this section the reader should recall that implicit methods were introduced as a means to describe the operational semantics of our model, independently of a specific implementation.

2.3.2 Field Redirection with Compound References

The central mechanism of this model is the notion of *compound references* (CR). In contrast to primitive references, the binding of a CR to an object is not absolute, but rather relative to another reference. To gain a first insight for the usefulness of CRs reconsider the account example from Sec. 2.2.1. The problem discussed there could be solved if it would be possible to express that “**person’s account**” - meaning the **account** reference within the context of the **person** reference - should be passed to the standing order processing unit. This is where CRs come into play.

A CR to a reference **instVar** within a class is created by means of **this<-instVar**. To illustrate their semantics, consider the class in Fig. 2.10⁶. The **getPersonsAccount()** method returns a compound reference to the **account** instance variable of a person, while **getAccount()** returns a primitive reference to the **account** instance variable of a person. The effect of the CR returned by **getPersonsAccount()** is that it always refers to the current value of the **account** reference within a **Person** object. After the **setAccount** call in the last statement of **Client::main** in Fig. 2.10, which changes Jack’s account from **ubsAccount** to **dbAccount**, **jacksAccount** will refer to Jack’s current “Deutsche Bank” account, while **anAccount** will still refer to his old UBS account. Fig. 2.11 and Fig. 2.12 schematically show the state before and after changing Jack’s account.

Just like object methods that differ from functions in the sense that different calls to them may return different values, depending on the state of the method’s owner (the receiver), a CR is different from a primitive reference in the sense that the evaluation of a CR might result in different values depending on the state of CR’s owner object. CRs are very different from pointers or pointers to pointers etc. A pointer always explicitly specifies the dimension of indirection (in C++ the number of *****). Pointer of different dimensions (for example, **Account **a1** and **Account ***a2**) are not compatible or substitutable⁷. CRs, on the other hand, are a transparent replacement for usual references: It is generally not known whether a reference is a CR or not, or how many levels of indirection are hidden in the

⁶Please note that in Java all object-typed instance variables are references.

⁷A *conversion* from ****** to ******* is possible in C++, for example **a1 = *a2**, but the semantics is different: If the first indirection of **a2** is changed after this assignment, **a1** still points to the previous account.

```

class Person {
  Account account;
  Account getAccount() { return account;}
  Account getPersonsAccount() {
    return this<-account;
  }
  void setAccount(Account newAccount) {
    account = newAccount;
  }
}

class Client {
  public static void main(String[] args) {
    Person jack = ...
    Account ubsAccount = new Account("UBS", "12345");
    Account dbAccount =
      new Account("Deutsche Bank", "54321");
    jack.setAccount(ubsAccount);
    Account anAccount = jack.getAccount();
    Account jacksAccount = jack.getPersonsAccount();

    // anAccount and jacksAccount
    // refer to the UBS account

    jack.setAccount(dbAccount);

    // anAccount still refers to the UBS account
    // but jacksAccount refers to the DB account
  }
}

```

Figure 2.10: Illustration of compound references

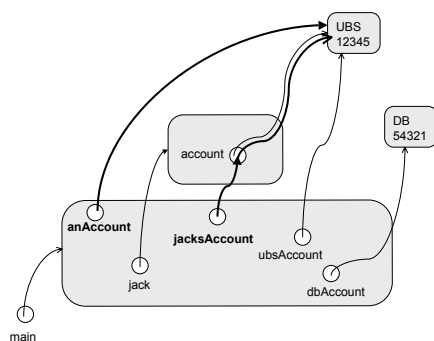


Figure 2.11: State before changing Jack's account

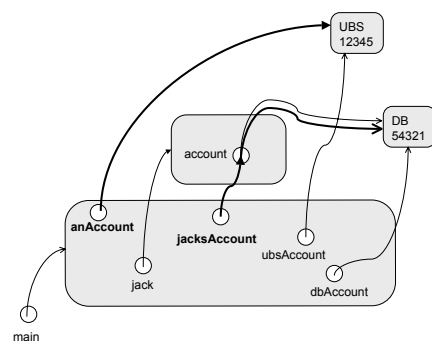


Figure 2.12: State after changing Jack's account

CR. In a way, CRs are similar to symbolic links in a Unix file system. A symbolic link may refer to a file or to another symbolic link. If objects were directories, one could create a symbolic account link in the `SOP` directory that refers to the account link in the `Jack` directory.

A CR can be defined relatively to a primitive reference or recursively to another CR. Hence, each CR may in general induce a *path* of object references. For example, a class `Person` might return a CR to the spouse of that person. If the `getPersonsAccount()` method on this CR is called, a new CR with path `personOID<-spouse<-account` is obtained. In general, a CR is an OID o together with a sequence of field names v_1, \dots, v_n . A CR $o<-v_1<-...<-v_n$ induces a corresponding path of objects $o_0<-o_1<-...<-o_n$ such that $o_0 = o$ and $o_i = o_{i-1}.v_i$ (details and subtleties about creating an object path for a CR are discussed in Sec. 2.5). Such a path is not created directly but incrementally as a result of creating a CR to a reference that is actually already a CR. In the following, a usual primitive reference is regarded as a special case of a CR of length one. Relative to an element o_i , o_{i-1} is called a *predecessor* and o_{i+1} a *successor*. Furthermore, o_0 is the *head* and o_n is the *tail* of the CR. Please note that a CR is itself immutable, while the corresponding object path may change in the course of time due to a changing instance variable on the path.

Just like any reference in a statically typed language, (a) a CR has a type, (b) it can be compared to other CRs, and (c) methods can be invoked on it. The *static type of a CR* is defined to be the static type of its tail, and the dynamic type of the tail is the *temporary type* of a CR, because the temporary type may change as a side effect of a field update. Downcasts to the temporary type of a CR are *disallowed* in this model because references that are typed to the temporary type may become invalid after a field update. This issue is further discussed in section 2.5.

Let us now consider the identity semantics in the context of compound references. The question is: Under which conditions are two compound references $s \equiv o<-v_1<-...<-v_n$ and $t \equiv p<-w_1<-...<-w_m$ with their corresponding object paths $o_0<-o_1<-...<-o_n$ and $p_0<-p_1<-...<-p_m$ considered identical?

There are at least three possible answers:

- **Head identity:** $s == t :\Leftrightarrow o == p$.
- **Tail identity:** $s == t :\Leftrightarrow o_n == p_m$.
- **Path identity:** $s == t :\Leftrightarrow n = m$ and $o_i == p_i$ for $i = 0, \dots, n$.

Head identity seems to be awkward because references would be considered identical that are - in general - not even of the same type (the static type of a CR is the static type of its tail). For example, CRs to the account resp. to the address of the same person would be considered identical. Path identity, on the other hand, seems to be too restrictive. Recall the account

```

class TextJustifier {
  private TreeIterator it;
  private void it.action(Object x) { ... }
  private boolean it.test(Object x) { ... }
  public void justify() {... it<-doAll(...); }
}

```

Figure 2.13: Explicit redirected field methods in TextJustifier

example. If Jack and Sally share an account, then one wants Jack’s account, i.e., the CR `jack<-account`, to be identical to Sally’s account, i.e., to the CR `sally<-account`. Hence, tail identity seems to be the only reasonable identity semantics. For this reason, two CRs are defined to be identical if and only if their tails are identical as defined above.

Finally, let us consider the method call semantics on a CR. If a method implemented by an object `o` is called via a CR on `o`, the value of the implicit `this` parameter is actually the CR and not `o`. That is, if during the execution of the method the object `o` passes itself to another object, it actually passes the CR by which the method was called. For convenience, some syntactic sugar is added: A method call `(this<-a).m()` is abbreviated to `a<-m()`.

For illustration, reconsider the `TextJustifier` implementation in Fig. 2.9. Truly incremental modification would mean to implement only `test()` and `action()` since these are the only methods, the semantics of which should be specific when used in the context of a `TextJustifier`. The question is now, how would then the specific iteration step semantics implemented by `TextJustifier::it.action()` get integrated into the iteration process which is performed by the `doAll()` method called on the instance variable `it` of a `TextJustifier`? Here is where the interplay between field methods and CRs becomes relevant. If methods are dispatched via a compound reference, field methods override corresponding methods of successive objects. In more detail, the semantics is as follows. Let `myref` \equiv `o<-v1<-...<-vn` be a CR with object path `o0<-o1<-...<-on` and `m()` be a method of the static type of `vn`. Furthermore, let i be the lowest index such that the class of `oi` contains a field method `vi+1....vn.m()` (a normal method is regarded as a field method with empty prefix). Then a method call `myref.m()` will be dispatched to the field method `vi+1....vn.m()`.

The implementation in Fig. 2.13 illustrates the interplay of CRs and explicit field methods. Within `TextJustifier::justify()`, the method `doAll()` is not called directly on `it`, but rather via the compound reference `this<-it`. Consequently, subsequent calls to `action()` and `test()` that are made within the control flow of `TreeIterator::doAll()` will be dispatched to `TextJustifier::it.action()`, respectively `TextJustifier::it.test()`.

With the implementation in Fig. 2.13, the fact that `TextJustifier` uses

```
// rf is a redirected field
private void rf.m() { field<-m(); }

// f is a non-redirected field
private void f.m() { field.m(); }
```

Figure 2.14: Field methods and field redirection

```
class TextJustifier {
    private redirect TreeIterator it;
    private void it.action() { ... }
    private boolean it.test(Item x) { ... }
    public void justify() {... it.doAll(...);}
}
```

Figure 2.15: TextJustifier with field redirection

and even customizes an `TreeIterator` is completely hidden from clients and subclasses of `TextJustifier`. It is not required for overriding methods to respect the visibility of the overridden methods, because `TextJustifier` is not a subtype of `TreeIterator`. Without any further code modification, it would also be possible to choose an iteration strategy at runtime (cf. subsection 2.2.3) by simply assigning a new iterator object to `it`. Thus, the implementation in Fig. 2.13 actually realizes a composition of `TextJustifier` and `TreeIterator` functionalities that supports overriding, transparent redirection, dynamic polymorphism, without subtyping and acquisition.

So far, compound references to an aggregated object referred to by a field `f` are only explicitly created by the aggregating class containing `f` before a method call (cf. `f<-m()`). That is, the scope of the composition features mentioned above (overriding, transparent redirection, and dynamic polymorphism) is an individual method call via an explicitly created CR to `f`. This is different with fields that are declared with the modifier `redirect`. Implicit field methods of a field that is annotated with the `redirect` keyword have a different semantics: Instead of simply forwarding the call to the field object, they first implicitly create a CR to that field and call the method on the created CR. Fig. 2.14 shows the difference between the default implementation of implicitly available methods of a redirected field, `rf` and that of a non-redirected field `f`. For illustration, a version of `TextJustifier` with `it` declared as a `redirect` field is given in Fig. 2.15.

2.3.3 Field Acquisition

Field acquisition is another step on the road from object composition to inheritance. Orthogonal to the other modifiers, the `acquire` modifier can

```
class OutputStream {
    public void write(int b) {
        System.out.println("Hello from write(int )");
    }
    public void write(int[] b) {
        System.out.println("Hello from write(int[] )");
    }
}

class EmptyFilterStream {
    acquire private OutputStream stream = new OutputStream();
}

class Client {
    static public void main(String[] args) {
        EmptyFilterStream efs = new EmptyFilterStream();
        int[] array = ...;

        efs.write(3);
        // "Hello from write(int )" appears

        efs.write(array);
        // "Hello from write(int[] )" appears
    }
}
```

Figure 2.16: `EmptyFilterStream` with acquired fields

also be a modifier of a field declaration. The intuitive semantics is that the features available in the field become an inherent part of the aggregating class. A class `C` with an acquired field `f` of type `F` implicitly contains a method `m()` for every public method `m()` of `F`. The method `m()` retains its signature as declared in `F` and its visibility in `C` is `public`. The semantics of the implicit field methods remains the same as with non-acquired fields, except that they are now provided in the interface of `C`. For illustration, consider the example in Fig. 2.16. Although `EmptyFilterStream` does not itself implement `write(int)` or `write(int[])`, these methods can be invoked on `efs` – an instance of `EmptyFilterStream` – due to the declaration of the instance variable `stream` as an acquired field.

Acquired implicit methods can also be replaced by explicitly programmed methods with the same signature. For the sake of uniformity and in order to facilitate changing of a given composition semantics by means of changing the modifiers of an instance variable, the prefix notation has to be used when explicitly overriding acquired methods. For illustration, consider the sample code in Fig. 2.17 and 2.18. The class `BufferedOutputStream` acquires both `write` methods from its acquired field `stream` and overrides them to add

buffering.

Note that the `bos.write(array)` call in the client code in Fig. 2.18 only displays “... buffering 5 ints ...”. The fact that no message “... buffering a single int ...” appears on the screen suggests that the overridden `write(int)` method as implemented in `BufferedOutputStream` is not invoked, although, at this point the `write(int)` method of the underlying `fout` stream will actually be called 5 times (since the buffer is already full, the overridden field method will be called for both the buffer and the int array passed as a parameter). However, “... buffering a single int ...” is not displayed because neither `stream` is a redirected field, nor are the calls `field.write(buffer)` and `field.write(b)` made via a compound reference `this<-stream`. Therefore, the calls to `write(int)` from within `field.write(...)` escape the override by the `BufferedOutputStream`. This corresponds to the *broken delegation problem* discussed in Sec. 2.2. In this case, this is indeed the desired semantics, i.e., redirection is actually not desired. Once the buffer is full, the buffer’s content should be flushed and the integers should be written immediately to the underlying data sink. Hence, buffering should indeed be escaped.

However, there might be cases, when one wants all calls occurring within the control flow of a call to an overridden acquired method of an object `outer` to be also dispatched to `outer`. If field acquisition is combined with field redirection, one obtains a perfect solution for this composition requirement. In the version of `BufferedStream` presented in Fig. 2.19, where `stream` is declared to be acquired and redirected, it suffices to do the buffering in the `write(int)` method, because calls to `write(int)` in the `write(int[])` method are automatically redirected to the buffering method.

One important restriction is imposed on field acquisition: Every class is allowed to have **at most one field acquisition**. Otherwise it would be necessary to take charge of all those annoying multiple inheritance conflicts. However, due to the fact that one can (a) do overriding and redirection for multiple fields, and (b) combine multiple classes by means of organizing them in an acquisition chain, this is no grave limitation.

There is a second restriction that needs to be imposed. Due to subtype polymorphism, an instance of a subtype of `OutputStream` may be assigned to the `stream` instance variable. This subtype may contain methods that are not available in `OutputStream`. These methods should not be overridden, because this might lead to unexpected or unsound results: The result might be unexpected because the author of the overriding method does not know about the existence and semantics of the overridden methods. The result might also be unsound, because the overriding method may have a signature that is not compatible with the signature of the overridden method (e.g., has a different return type, see also [Kni99]). For this reason, the following restriction is necessary: A field method overrides a method defined in a field type if and only if it is already defined in the *static* type of the field type.


```
class OutputStream {
    public void write(int b) { ... }
    public void write(int[] b) {
        for (i = 0; i < b.size(); i++) write(b[i]);
    }
}

class BufferedOutputStream {
    acquire private OutputStream stream;
    int[] buffer; int current;

    public BufferedOutputStream(out) {
        stream = out;
        buffer = ... ;
    }

    public void stream.write(int b) {
        System.out.println("... buffering
                           a single int ... ");
        if (buffer.notFull()) buffer[current++] = b;
        else {
            field.write(buffer);
            field.write(b);
            current = 0;
        }
    }

    public void stream.write(int[] b) {
        System.out.println("... buffering "
                           + b.size + "ints ...");
        if (buffer.size() >= current + b.size()) {
            System.arraycopy(b,0,buffer,current,b.size());
            current += b.size;
        } else {
            field.write(buffer)
            field.write(b);
            current = 0;
        }
    }
}
```

Figure 2.17: Overriding acquired fields

```
class Client {
    static public void main(String[] args) {
        FileOutputStream fout =
            new FileOutputStream(aFileName);
        BufferedOutputStream bos =
            new BufferedOutputStream(fout);
        int[5] array = ...;

        bos.write(3);
        // "... buffering a single int..."
        // appears on the screen
        ...

        //assume that buffer is full at this point

        bos.write(array);
        // "... buffering 5 ints ..."
        // appears on the screen
    }
}
```

Figure 2.18: Client code for Fig. 2.17

```
class BufferedStream {
    acquire redirect private OutputStream stream;
    public void stream.write(int b) {
        ...do buffering...
        field.write(b);
    }
}
```

Figure 2.19: Overriding and redirecting acquired fields

The addition of the **acquire** feature into the model, has further enriched the range of composition semantics between the classes **C** and **F** that the programmer can express. Used in isolation, **acquire** enables **C** to transparently forward services to **F**, whenever it needs to do so, in order to satisfy a request from an external client. On the other hand, combining **redirect** and **acquire** yields a mechanism for incremental modification that mimics the code reuse provided by inheritance or delegation.

2.3.4 Subtyping

One thing is still missing on the road to inheritance/delegation-based composition: Field acquisition does not imply subtyping. A class can explicitly declare to be a subtype of a number of other types via a **subtypeof** clause.

```
class OutputStream { ... }
class FileOutputStream extends OutputStream { ... }

class FilterStream subtypeof OutputStream {
    acquire redirect protected OutputStream stream;
}
class BufferedOutputStream extends FilterStream {
    public void stream.write(in b) {
        ... do buffering ... ;
        field.write(b);
    }
}
class CompressedOutputStream extends FilterStream {
    ...
}
```

Figure 2.20: OutputStream in our model

Declaring a class *C* as a subtype of a type *T*, requires that *C* has to either implement all methods that are defined in *T* or be abstract. In contrast to Java's `implements` clause, in our model both interfaces *and* classes may appear on the right hand side of a `subtypeof` clause⁸.

Declaring a class *D* to be a subtype of another class *C* means that *D* implements the interface of *C*, but it does not mean that the implementation of *C* can automatically be used for the realization of the corresponding methods in *D* - *D* does not automatically acquire the state and method implementations of *C*. However, *D* can still make use of the behavior defined in *C*, if this is desired, by declaring a field of type *C* with modifiers `acquire` and `redirect`. This is an important step towards a better separation of types and classes. Decoupling subtype declaration from implementation reuse solves e.g., the last drawback of the decorator approach explained in section 2.2.2.

For illustration, the complete implementation of the stream example from Fig. 2.2 in our model is given in Fig. 2.20. Compare this to the simulation of redirect semantics in Fig. 2.3.

2.3.5 Field Navigation

If the object referred to by a field represents a facet that should be visible to clients, it is desirable to make this fact explicit in the declaration of the field, rather than relying on the presence of appropriate getter methods. For serving this purpose, a field can be made navigable by annotating it with the `navigable` modifier. For example, one could annotate the `account` field of `Person` as navigable, as shown below. This allows clients to directly navigate to this part of the object by retrieving a compound reference to that part.

⁸It is still possible to use traditional inheritance with `extends`.

This is illustrated below by having the client of the `Person` object `p` retrieve a CR to `p`'s `account` and store it in `a`. Technically, the declaration of a field as navigable can be seen as short-cut for the corresponding getter methods discussed above.

```
class Person {
  private navigable Account account;
}
Person p = ...
Account a = p<-account;
```

Note that declaring a field as navigable does not imply that clients can directly change the field. The possibility to navigate to a field becomes part of the class interface, similar to a getter method that returns the current value of a field. Actually, the navigable composition semantic flavor discourages rather than supports breaking encapsulation. Exporting a CR to an instance variable to external clients as part of the interface of a class `C` can also be simulated by declaring a redirect field to be public. However, this breaks the encapsulation of `C`: clients can freely change the value of the reference. This is not possible with navigable references.

The inverse navigation operation is provided by a CR *reduction operator* that can be used to access previous objects on the object path of a CR. A reduction `<AType> myref` on a compound reference `myref \equiv o<-v1<-...<-vn` creates a new compound reference `o<-v1<-...<-vn-1`. The reduction succeeds if the type of the shortened compound reference (that is, the static type of the `vn-1` variable) is a subtype of `AType`. If the length of the source path is two, a primitive reference to `o` is created. Since it is not statically known whether an account reference is a compound reference via a person, a CR reduction has to be checked at runtime. The reduction operation is in a way similar to type downcasts in languages like Java. In the `Account` example, a compound reference to an `Account` could be reduced to `Person`:

```
Person p = ...
Account a = p<-account;
Person p2 = <Person> a; // ok, checked at runtime
(p == p2) // true
```

2.4 Evaluation of the Model

After having introduced the individual steps on the road from object composition to inheritance, it is now time to show how the problems discussed in Sec. 2.2 are addressed in our model. The key to addressing these problems is the availability of rich linguistic means to express a variety of composition flavors by simply decorating object references with composition properties. To support the discussion, Fig. 2.21 introduces graphical notations for some

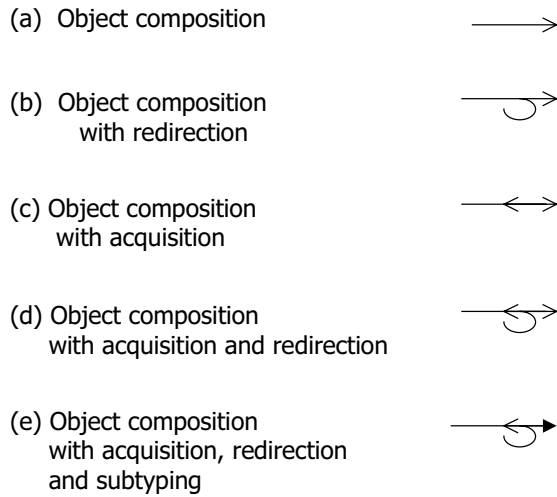


Figure 2.21: Graphical notations for different composition flavors

of the most relevant composition flavors between two classes **C** and **F** that can be expressed with the model⁹. Please note that these notations do not address overriding because in our model overriding is implicitly available by means of field methods and need not be explicitly turned on or off.

Let us start with the the composition flavor (b) - in the middle of the road between object composition and inheritance. A composition **C(F)** with this flavor shares with inheritance overriding with late binding. This is not available with object composition. On the other side, such a flavor shares with object composition dynamic polymorphism as well as lack of both acquisition and subtyping. The latter two features are inseparable from inheritance/delegation, though. The discussion of the **TextJustifier** example in Sec. 2.2 indicated that such a mixture of features might indeed be needed and that the lack of linguistic means to express it forces the programmer in an object-oriented language to simulate the same semantics by means of complex, unclear architectures that are fragile with respect to requirement changes.

In the previous section, the same composition scenario has been described in our model. The implementation of the desired composition semantics between **TextJustifier** and the **TreeIterator** hierarchy is as simple as the code in Fig. 2.15 and the design as clear as the class diagram presented in Fig. 2.22, which is as simple as the inheritance-based design in Fig. 2.5.

⁹This list is not intended to cover all possible combinations of composition features, but only those that are relevant for evaluating the model with respect to the issues discussed in Sec. 2.2.

However, in contrast to the inheritance-based design, with the CR-based solution (1) several iteration strategies can be chosen dynamically, (2) the iteration functionality does not pollute the interface and implementation of `TextJustifier`, and (3) the conceptual view that a `TextJustifier` is not a special kind of `TreeIterator` is preserved. The latter are features that were indeed also supported by the architecture based on object-composition presented in Fig. 2.6 and 2.7. However, our design does not share the complexity of the designs presented in Fig. 2.6 and Fig. 2.7.

The complexity of designs that simulate non-standard composition flavors was only one of the problems that have been identified in Sec. 2.2. The second and more important problem was that different composition flavors were modeled by different architectures. In the following, it is demonstrated that this problem can be avoided in this model, by reconsidering the text justifier and stream example from Sec. 2.2.

The design in Fig. 2.22 encodes a composition with redirection, overriding and dynamic polymorphism. Assume that acquisition would also be required. In this model, one would simply add the `acquire` modifier to the declaration of `it`. The class diagram in Fig. 2.22 remains the same, except for replacing the current `it` link with link (d) in Fig. 2.21. On the contrary, with the designs in Fig. 2.6 and Fig. 2.7 one would have to change `TextJustifier` to implement all methods in the interface of `TreeIterator` by forwarding these methods to `it`. If one additionally wants to have `TextJustifier` be a subtype of `TreeIterator`, it would again merely be necessary to replace the `it` link with the link (e) in Fig. 2.21. The resulting design would still encode a different composition flavor as compared to the inheritance based composition in Fig. 2.5, since (1) one still has a composition that supports dynamic polymorphism and (2) `TextJustifier` would not inherit the state of `TreeIterator`.

A similar seamless transition from one composition flavor to the other was observed when different flavors of `BufferedOutputStream` in Fig. 2.17 and 2.18 have been modeled, see Fig. 2.19 and Fig. 2.20. Here we started with a flavor that is closer to the inheritance end of the composition flavor spectrum: Object composition with acquisition semantics. Then redirection and subtyping have been added in two separate steps.

Another important feature of our model which makes it superior to standard composition models is the fact that a class can simultaneously reuse and adapt the functionality of several other classes without suffering from the known multiple-inheritance conflicts:

- Naming conflicts: Different methods with the same name are inherited.
- Repeated inheritance (a.k.a. diamond inheritance): The same class is inherited twice indirectly, for example D is a subclass of B and C, both of which are subclasses of A. Is a single copy of A shared by B and C or are there two copies? What happens to methods that are overridden

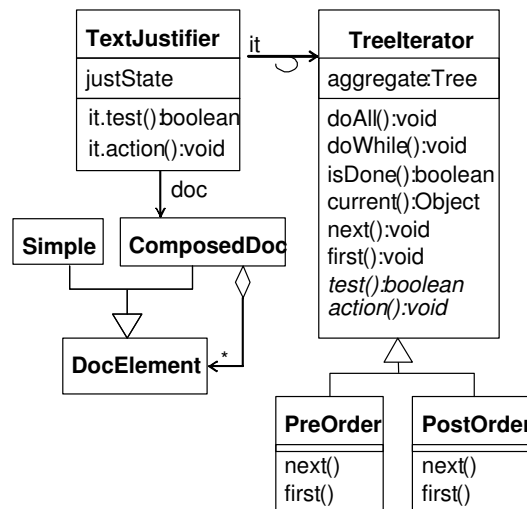


Figure 2.22: TextJustifier using compound references

in B and C in the first case? Which copy of A do clients of D see in the latter case?

Different mechanisms have been developed to cope with these problems, but avoiding a problem is certainly better than fixing it. Due to the naming definition for field methods (prefixing it with the name of the attribute), there are no naming conflicts. Problems related to repeated inheritance do not occur either, because every compound reference induces a unique path for message dispatch. The question whether one has a shared or replicated parent boils down to assigning the same, respectively different instances of the aggregated class to the corresponding attributes.

Finally, navigable fields should be considered in this evaluation, because they foster another spectrum of relationships not discussed so far. Industrial component models like COM [Box97] and CCM [OMG99] have the notion of independent interfaces or facets that a component exposes and that can be retrieved by special navigation methods. Design patterns such as *extension interface* [SSRB00] or *extension object* [Gam98] propose architectures to allow a class to export multiple unrelated interfaces à la COM and CCM without employing inheritance or subtyping. However, as also acknowledged by the authors, the proposed patterns incur increased design and implementation effort, e.g., navigation infrastructure that is of no functional use but necessary to retrieve the facets [SSRB00], and increased client complexity [Gam98, SSRB00]. This critique is in the vein of our discussion in the motivation section.

Navigable fields present an elegant approach to modeling classes that export several unrelated interfaces. A class C exports the interfaces of all

navigable fields. This is explicitly declared in the class' interface. This export involves no interface bloat because *C*'s interface does not itself contain the methods of the exported interfaces. This is in contrast to a class in Java implementing several different interfaces. In contrast to the extension interface and extension object patterns, the feature of exporting several unrelated interfaces is built into the language and integrated with static type checking. The relationship that the exported interfaces are facets of the behavior of the exporting class is explicit in the exporting class' interface. The same relationship is not explicit in the design of the extension interface and extension object patterns as also indicated by Gamma [Gam98].

2.5 Reconciling dynamic specialization and static typing

In this section, we will elaborate on type safety issues related to compound references. In particular, a method dispatch strategy is presented that is statically type safe. Type safety is threatened by subtle combinations of compound references and subtype polymorphism.

In section 2.3.2, the static and the temporary type of a CR have been defined. It was argued that type conversions to the temporary type of a CR should not be allowed because the temporary type may change in the course of time. Enforcement of this invariant is trivial for explicit type conversions (casts) in the program code. However, there are situations when type conversions of a CR to its temporary type are inevitable: Consider a class *A* with a field *b* of type *B*. At runtime, an instance of *BSub*, a subtype of *B*, is assigned to *b* and *A* makes a call *b*←*m*() to this object. The method *m*() of *B* is overridden in *BSub*. This means that method *m*() of *BSub* is executed and the actual value of *this* is the CR *a*←*b* with static type *B*. However, the type of *this* has to be (at least) the type of the enclosing class *BSub* because otherwise features that are introduced in *BSub* could not be called. We call this kind of inevitable type conversion an *implicit conversion to the temporary type of a CR*.

These casts to the temporary type are the cause that under certain conditions the naive algorithm for creating an object path $o_0 \leftarrow o_1 \leftarrow \dots \leftarrow o_n$ for a CR $o \leftarrow v_1 \leftarrow \dots \leftarrow v_n$, namely $o_0 = o$ and $o_i = o_{i-1}.v_i$, fails.

Fig. 2.23, 2.24, and 2.25 show three different scenarios that threaten type safety if the aforementioned naive algorithm is employed. The problem in all cases is the storage of a critical CR. Critical CRs can be stored in three different places:

- **In a usual object reference.** This is the scenario shown in Fig. 2.23. The object *o* stores a critical CR which is assumed to be of type *BS*, an assumption that is invalidated by the update of *a.b* with an instance *b* of the class *B*.

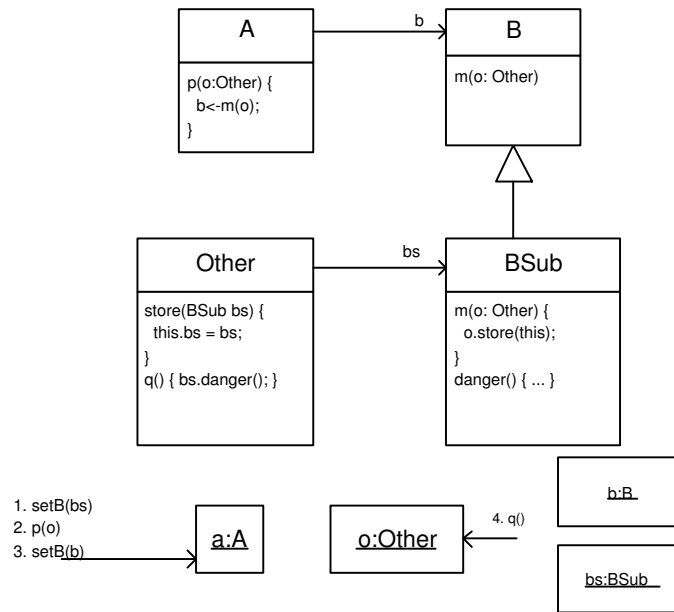


Figure 2.23: Type safety problem: **Other** assumes that **bs** has type **BSub**

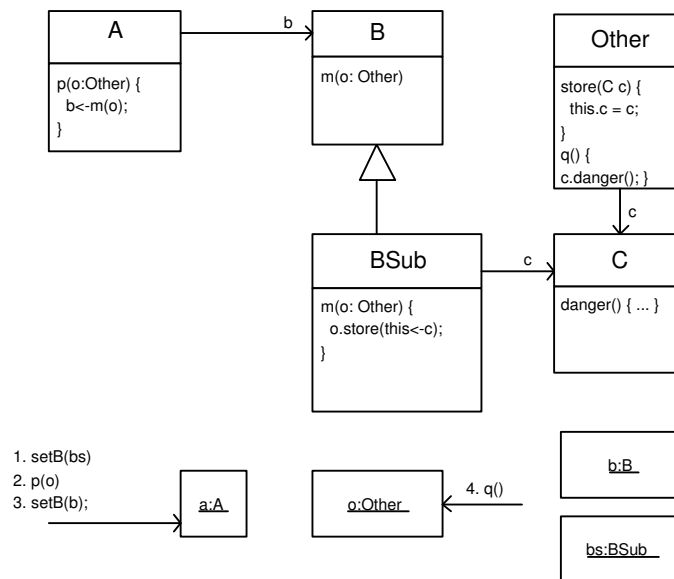


Figure 2.24: Type safety problem: CR path cannot be evaluated because **B** has no field **c**

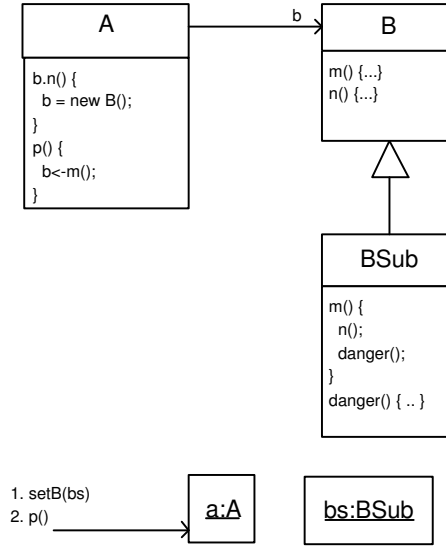


Figure 2.25: Type safety problem: Updating while a critical CR is still on the callstack

- **Inside a CR that includes the critical CR.** This is the scenario shown in Fig. 2.24. The CR that is created inside `BSub.m()` and stored in `o` includes the critical CR to `BSub`. After an update of `a.b`, the CR cannot be evaluated because `B` has no field `c`.
- **On the callstack.** This is the scenario shown in Fig. 2.25. The field `a.b` is updated while the method `BSub.m()` is still on the callstack. The call to `danger()` cannot be dispatched to the current CR target `b` because `B` has no `danger()` method.

Similar problems also occur in statically typed languages that feature delegation. Different solutions to cope with this problem have been proposed. Kniesel [Kni00, Chap. 5] classifies these solutions as follows:

- **Pure Specialization.** Pure specialization means that the new value of a “parent” reference has to be a subtype of the previous value. This solution is too restrictive and not applicable in our model. It is too restrictive because there are many useful scenarios within which the new “parent” is not a subtype of the previous one. In addition, this restriction cannot be enforced statically offhand, hence dynamic checks are needed to enforce it. This solution is also not practicable for our purposes because, in our model, *every* field is a potential target of a critical CR, and not only designated “parent” fields.
- **Waiting for future states.** This means that if a type error would occur in the current state, the current thread is halted until the method

can be correctly dispatched. This solution is clearly impracticable because it implies indefinite waiting (in a single-thread system: endless waiting).

- **Invariant split self.** In this approach, the usage of an object as parent needs to be anticipated during the design of the parent class. In potential parent objects, the `this` reference is split into two different pseudo-variables, `receiver` and `holder`. Only calls to `receiver` are late-bound to potential child objects. Type-safety is preserved by the restriction that the type of `receiver` does *not* change covariantly in subclasses, see [Kni00] for more details. This solution is semantically clean and safe, but it does not fit to our programming model because *all* objects might be the target of a critical CR, hence an anticipation of potential CR targets is neither acceptable nor feasible.
- **Frozen paths.** The rationale behind this approach is that previous parents are stored such that under certain conditions (in particular: if a type error would occur otherwise) methods are dispatched to the stored old parents. The advantage of this approach over the other ones is that it does not restrict the programming model. A potential disadvantage is that the semantics of dispatching methods to outdated parents may be confusing or semantically questionable.

The frozen path idea is the only feasible approach that preserves static type safety without restricting the programming model. In the following, we will present a dispatching strategy in this solution class. Its main difference over the frozen path approaches presented in [Kni00] is that the target of a method call does not depend on the *method name* but on the *static type* of the variable that is used to call the method. We think that this leads to a cleaner and easier semantics because otherwise the object path that is created may be different for every method. In our case, it is guaranteed that all methods to a CR that are called via the same variable are dispatched along the same object path.

The main idea of this approach is as follows: On every implicit conversion to the temporary type of a CR, the current field value is stored in the CR. This original value is used whenever the current value would lead to a type error in the sense that the current tail of the CR is not a subtype of the static variable type.

A CR that has been implicitly converted to its temporary type (or a supertype of the temporary type that is not a supertype of the static type) is called a *critical CR*. For the definition of this algorithm, a recursive representation of CR is used: A CR is either a primitive reference *primRef*, or a non-primitive compound reference *parent*<-*v*, whereby *parent* is a CR and *v* a field name, or a critical compound reference *parent*<-*v* | *s*, whereby *s* is the stored field value. Please note that due to the recursive construction

parent may already be a critical CR. A non-critical CR is converted to a critical CR by storing the current value of the field *v* in *s*. This conversion (think of the CR as being passed *by value*) takes place whenever the CR is subject to an implicit conversion to its temporary type. For example, in Fig. 2.23, the CR *a*←*b* that is created inside *A.p()* is converted to a critical CR in the scope of the execution of *BSub.m()* because the CR needs to be implicitly converted to its temporary type in order to execute *BSub.m()*. The stored field value *s* would in this case be the current value of the CR, *bs*.

The decision whether the current or the stored field value is used is based on an additional parameter, the *requested type reqType*, that defines which type is expected in the actual context. For a CR *ref*, *reqType* is the static type of *ref*. In the following, C_v denotes the class in which the field *v* is defined. For primitive and non-critical CRs, the object path is created as follows:

$$\begin{aligned} \text{objectPath}(\text{primRef}, \text{reqType}) &:= \text{primRef} \\ \text{objectPath}(\text{parent} \leftarrow v, \text{reqType}) &:= \\ &\quad \text{objectPath}(\text{parent}, C_v) \leftarrow \text{tail}(\text{parent} \leftarrow v, \text{reqType}) \\ \text{tail}(\text{primRef}, \text{reqType}) &:= \text{primRef} \\ \text{tail}(\text{parent} \leftarrow v, \text{reqType}) &:= \\ &\quad \text{tail}(\text{parent}, C_v).v \end{aligned}$$

Except for the additional *reqType* parameter, this algorithm is equivalent to the non-recursive description $o_i = o_{i-1}.v_i$: The object path of a primitive reference is the primitive reference itself, and the object path of a compound reference is the object path of the “parent” CR concatenated with the current tail value.

In the critical case, the *reqType* parameter comes into play:

$$\begin{aligned} \text{objectPath}(\text{parent} \leftarrow v \mid s, \text{reqType}) &:= \\ &\quad \text{objectPath}(\text{parent}, C_v) \leftarrow \text{tail}(\text{parent} \leftarrow v \mid s, \text{reqType}) \\ \text{tail}(\text{parent} \leftarrow v \mid s, \text{reqType}) &:= \\ &\quad \text{if } \text{tail}(\text{parent}, C_v).v \text{ instanceof } \text{reqType} \\ &\quad \quad \text{then } \text{tail}(\text{parent}, C_v).v \\ &\quad \quad \text{else } s \end{aligned}$$

If the tail of a critical CR is evaluated, a typecheck *instanceof* decides whether the current tail value or the stored tail value is used. For illustration, let us consider the evaluation of *bs.danger()* inside *Other.q()* in Fig. 2.23. The *this* reference that is passed to *o* in the *o.store(this)* call inside *BSub.m()* is a critical CR, whereby the stored field value points to *bs*. Since the requested type during the evaluation of *bs.danger()* is

`BSub` (the static type of `Other.bs`), the `instanceof` test in the definition of `tail(parent<-v | s, reqType)` fails and `s` (the reference to the `BSub` instance `bs` in this case) is returned. In Fig. 2.24, the critical situation occurs in the evaluation of `c.danger()` inside `Other.q()`. In this situation, the value of C_v in the recursive call to `objectPath` is `BSub` because `c` is defined in `BSub`. Hence, the subsequent call to `tail` yields the stored field value again, which points to `bs`. The call `c.danger()` is hence dispatched to `bs.c`. The situation in Fig. 2.25 is similar to Fig. 2.23, except that this time the critical CR is on the callstack: Since the static type of `this` inside `BSub.m()` is the enclosing class `BSub`, the `reqType` parameter is `BSub` and the stored value `s` is returned in the `this.danger()` call inside `BSub.m()`.

An induction proof on the length of CR shows that this algorithm preserves type safety. It suffices to proof that the type of the object that is returned by the `tail` function is always a subtype of `reqType`. For CRs of length one (that is, primitive references) the claim holds because the base language (without CR) is assumed to be statically type safe. Let `ref` be a CR with length n and static type T . Then `ref` may be a critical or a non-critical CR.

1. **ref is non-critical.** Let `ref` \equiv `parent<-v`. $T = reqType$ is a supertype of the static type of `ref` because otherwise `ref` would be critical. By induction hypothesis, the type of `tail(parent, C_v)` is a subtype of C_v , so that the field value `v`, `ov` can be safely retrieved. Subtyping guarantees that this object is a subtype of the static type of `v`, and therefore also a subtype of `reqType`.
2. **ref is critical.** Let `ref` \equiv `parent<-v | s`. By induction hypothesis, the type of `tail(parent, C_v)` is a subtype of C_v , so that the field value `v`, `ov` can be safely retrieved. The `if` statement guarantees that `ov` is returned if and only if it is an instance of `reqType`. If this is not the case, `s` is returned, so it is necessary to show that `s` is a subtype of `reqType`. This is assured by the rule that a critical CR is created and initialized whenever a non-critical CR is implicitly converted to its temporary type. Hence, during the assignment of `ref`, `s` has been assigned to a subtype of the static type of `ref`.

The dispatch algorithm presented here renders the model statically type-safe. However, a price has to be paid: It is unfortunate that the invariant “all calls to a CR are always dispatched to the current object of the corresponding field” does not hold anymore. It would be desirable to have a better strategy that does not violate this invariant but is still unrestrictive. A possible idea would be to detect whether there are critical CRs that point to the target of a field whenever the field is written. If field writes would be prevented in such situations, the typing problem would go away. However, it is unclear whether it is possible to detect such a dynamic situation

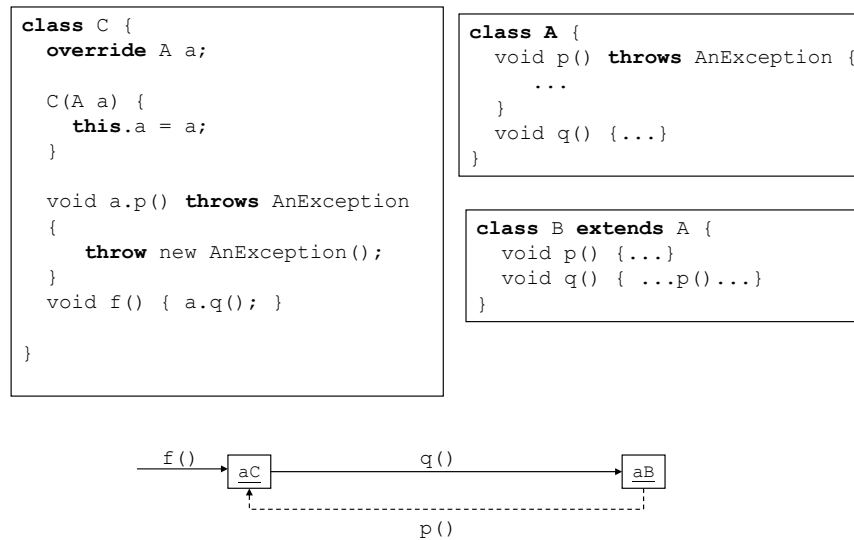


Figure 2.26: Unsafety of method header specializations: $B::q()$ assumes that $p()$ will not throw an exception

appropriately with a conservative static algorithm.

2.6 Abstract Classes and Method Header Specializations

Subtype polymorphism is usually defined in terms of subsets of available methods: an object type A is a subtype of B , if A has all features of B and possibly more. This definition of subtyping can be insufficient if it is employed in a scenario where methods of objects that are only known by upper bound can be overridden, as with delegation or our compound reference model. For example, this is the case if one considers abstract classes and method header specializations:

- **Abstract classes:** Abstract classes are an important conceptual mechanism to force overriding of methods. It would be useful to transfer this concept to dynamic overriding, that is, allow instances of abstract classes to be used as values for overriding attributes. The purpose of this mechanism would be the same as in the static case: To force overriding of methods. However, in the dynamic case we have to deal with the complex situation that we might have instances of abstract classes. Means to prevent unsafe method calls to an instance of an abstract class are needed.

- **Method header specializations.** An overriding method is said to specialize the header of its “parent” method, if it widens the domain of the method or narrows its range. In Java, for example, an overriding method can declare less exceptions than the method it overrides, or it can declare a method as final that is non-final in the superclass. Other examples of method header specializations are covariant return type definitions and contravariant parameter redefinitions. A combination of method header specialization with dynamic overriding may lead to problems, however. Fig. 2.26 shows a class *C* that overrides the method *p()* of a field *a* that is only known by upper bound *A*. *B* overrides *p* with a method header specialization (it omits the exception *AnException*) therefore the call to *p()* inside *B.q()* does not need to be guarded by a *try..catch* block. The problem is that *p* is also overridden in the context of *C*, this time without omitting the exception. If *a.p()* overrides *B.p()*, there is no exception handler for the exception that is thrown in *a.p()*.

The crucial point is that substitutability for usual clients does not imply substitutability for clients that may also override methods of the object. For this reason, an extended type system is presented that takes the aforementioned observation into consideration.

In the following definitions, we call an object *a* that calls methods of another object *b* without overriding methods of *b* a *client* of *b*. If *a* overrides methods of *b*, we call *a* a *specialization client* of *b*. The basic idea is that we introduce a more restrictive subtyping relation for specialization clients by means of *specialization interfaces*. For each type *X* we define *@X* to be the *specialization interface* of *X*. The subtyping relation of specialization interfaces does not only consider sets of available methods but also additional informations that are of interest to specialization clients, for example informations about final or abstract methods. The subtyping definition is as follows:

$$A \text{ subtypeof } B \quad :\Leftrightarrow \quad \text{publicMethods}(B) \subseteq \text{publicMethods}(A) \quad (2.1)$$

$$\begin{aligned} @A \text{ subtypeof } @B \quad :\Leftrightarrow \quad & A \text{ subtypeof } B \quad \wedge \\ & \text{abstractMethods}(B) \supseteq \text{abstractMethods}(A) \quad \wedge \\ & \text{no method header specializations in } A \end{aligned} \quad (2.2)$$

$$@A \text{ subtypeof } A \quad :\Leftrightarrow \quad A \text{ interface or } A \text{ non-abstract class} \quad (2.3)$$

$$A \text{ subtypeof } @A \quad :\Leftrightarrow \quad \text{false} \quad (2.4)$$

The first definition is the usual subtype polymorphism. The second definition ensures that a subtype of a specialization interface does not contain method header specializations or additional abstract methods. The third definition ensures that an instance of an abstract class can never be assigned to a client interface. The fourth definition states that a usual

interface is never a subtype of a specialization interface. The missing definitions follow from the transitivity of the subtyping relationship, for example $@A \text{ subtypeof } B \Leftrightarrow @A \text{ subtypeof } A \wedge A \text{ subtypeof } B$.

The extended type system is integrated into the language by the following definitions:

- The type of the reference of a specialization client is required to be a specialization interface.
- Method calls `instVar.m()` on an instance variable `instVar` with type `@X` are allowed if and only if `@X subtypeof X`.
- For any class `C`, the type of the expression `new C()` is `@C`.
- For an instance variable `x` of type `@X` in a class `C`, the type of `this<-x` is `X`.

An example for these mechanisms can be found in figure 2.27. The abstract class `IteratorImpl` can be safely instantiated and used as a value for the `it` attribute in `TextJustifier`. The non-abstract class `SafeIteratorImpl` cannot be used for this purpose because it contains method header specializations. Nevertheless, it can be assigned to a variable of type `Iterator` because the client interface of `SafeIteratorImpl` is a subtype of the client interface of `Iterator`.

2.7 Related Work

Delegation appeared first in untyped, prototype-based languages [Lie86]. The most prominent example in this category is `SELF` [US87]. As shown in Tab. 2.1, delegation includes all composition properties simultaneously; applying individual properties independently is not explicitly supported.

More recent proposals have been proposed to restrain the extreme flexibility offered by `SELF` and a number of related proposals by embedding delegation in a statically typed language. The `DARWIN` model [Kni99, Kni00] combines delegation and static inheritance in a statically typed language. `DARWIN` already incorporates a limited variant of composition property separation: Besides delegation and inheritance, `DARWIN` also has the notion of *consultation*, which, in our terminology, corresponds to delegation without redirection.

`GENERIC WRAPPERS` [BW00] support a restricted variant of delegation: Once a “wrappee” is assigned to a “wrapper”, the wrappee is fixed. In our terminology, this corresponds to delegation with “semi-dynamic” polymorphism (parent fixed at runtime), and in our model would be expressed by declaring the corresponding attribute as `final`. Büchi and Weck [BW00] emphasize the importance of being able to dynamically cast a wrapper to


```
abstract class Iterator {
    protected abstract void action(Item x);
    public void doAll()
        throws IterationException {}
}
class TextJustifier {
    redirect protected @Iterator it;
    public TextJustifier(@Iterator it) {
        this.it = it; }
    protected void it.action(Item x) {...}
}
abstract class IteratorImpl implements Iterator {
    protected abstract void action(Item x);
    public void doAll() throws AnException {...}
}
class SafeIteratorImpl implements Iterator {
    public void action(Item x) {
        defaultAction(it);
    }
    public void doAll() { // throws no exception
    }
}
@IteratorImpl i1 = new IteratorImpl();
Iterator i2 = i1;    // static error, no subtype
i1.action();        // static error, no direct calls allowed
new TextJustifier(i1);    // ok, @IteratorImpl subtype of @Iterator
@SafeIteratorImpl i3 = new SafeIteratorImpl();
Iterator i4 = i3;    // ok, @SafeIteratorImpl subtype of Iterator
new TextJustifier(i4);    // error, @SafeIteratorImpl
                        // no subtype of @Iterator
```

Figure 2.27: Example for subtyping of specialization interfaces

the dynamic type of its wrappee (*transparency*). In our model, this could be achieved by allowing explicit dynamic casts to the temporary type of a CR, which is not problematic, when the attributes are annotated `final`. However, further details on this aspect have been left out of the scope of this model.

GBETA [Ern99] also has a number of dynamic features that are related to delegation. Like in GENERIC WRAPPERS, parents in GBETA are fixed at runtime. GBETA also allows dynamic behavior additions to objects that preserve object identity, for example a statement like `aClass##->anObject##` adds the structure of `aClass` to `anObject`. Another delegation-based approach is described in [SM95]. Steyaert and De Meuter propose a variant of delegation in which a class has to anticipate all its possible extensions in order to avoid certain encapsulation problems.

Compared to these approaches to supporting delegation in a statically typed language, delegation, in our model, comes out as a special mixture of composition properties, among many other possible mixtures. In addition, our model is more flexible in that in contrast to the aforementioned approaches, objects do not have a *single special* parent attribute. In our model it is possible to override and redirect *multiple arbitrary* attributes.

PREDICATE OBJECTS [Cha93], RONDO [Mez97], and the CONTEXT RELATIONSHIP [SPL98] allow the programmer to express certain kinds of context-dependent facets of an object by explicit linguistic means. The composition of the basic behavior of an object and its facets obeys delegation semantics in [Cha93, SPL98], and some form of mixin-based inheritance in [Mez97]. Our model shares with these approaches the support for a two-dimensional incremental modification: (1) vertically by means of inheritance in our model, RONDO, and CONTEXT RELATIONSHIP, respectively by means of delegation in PREDICATE OBJECTS, and (2) horizontally by means of an advanced form of delegation in [Cha93], an advanced form of inheritance that supports the static/dynamic polymorphism property in [Mez97, SPL98], and by means of **CRs** in our model. However, in [Cha93, Mez97, SPL98] the composition flavors in both axes are built in; individual composition properties are not explicitly available for on-demand combination.

MIXIN-BASED INHERITANCE [BC90, FKF98, ALZ00] is an enrichment of normal inheritance with the static polymorphism feature of Tab. 2.1. In contrast to the normal inheritance, the **super** pseudo-variable of a subclass is not bound to a certain base class when the subclass is defined. Rather, there is an explicit composition stage, where **super** is statically bound to a composition-specific superclass. In this way, the same subclass (*mixin*) can be statically applied to several base classes. However, inheritance enhanced with static polymorphism is the only composition flavor supported.

JIGSAW [BL92] improves the modularity of the original mixin-based inheritance [BC90] by providing a suite of language operators that independently control several roles that classes play in standard languages such as combination of features, modification, encapsulation, name resolution, and sharing. This untangling of class composition semantics is in its core very similar to our untangling of standard composition semantics. The motivation for undertaking these untanglings is different, though. The main focus in JIGSAW is on fine-grain control over the visibility of the features from the individual modules in a composition, to allow mixins, multiple inheritance, encapsulation, and strong typing to be combined in cohesive manner.

The flexible control over the method dispatch via filters attached to an object complemented by the ability to define different facets of an objects in so-called **internal** and **external** objects supported by the COMPOSITION FILTERS approach [AWB⁺93] can probably also be used to simulate some of the flavors of composition semantics that can be expressed in our model. Still, there are important differences between the two models. First, the

COMPOSITION FILTERS approach lacks a static type system. Second, different flavors of composition semantics need to be manually implemented in different dispatch filters. This might turn out to be a tedious and error-prone activity, especially if several internals and mixtures of composition properties are involved. In contrast, the specification of the desired semantics is more declarative in our model. Third, it is not obvious how redirection semantics could be "programmed" with dispatch filters.

Our notions of field methods and field navigation share some commonality with the **as-expressions** of the POINT OF VIEW NOTION OF MULTIPLE INHERITANCE [CG90] in that they allow to adapt and combine multiple classes without suffering from multiple inheritance conflicts. However, due to the use of object rather than class composition, our approach is more flexible when coping with issues such as sharing and duplicating the features of common parents, typical for approaches to multiple inheritance.

2.8 Chapter Summary

In this chapter, it has been showed that the traditional object-oriented composition mechanisms, object composition and inheritance/delegation, are frequently inappropriate to model non-standard composition scenarios. Non-standard composition semantics are simulated by complicated architectures that are sensitive to requirement changes and cannot easily be adapted without invalidating existing clients. This unsatisfactory situation is due to the fact that the combination of composition properties supported by each mechanism is fixed in the language implementation and individual properties do not exist as abstractions at the language level.

Compound references have been proposed as a new and powerful abstraction for object references. On this basis, it was possible to provide explicit linguistic means for making individual composition properties available and to allow the programmer to express a seamless spectrum of composition semantics in the interval between object composition and inheritance. The model is statically type-safe and makes object-oriented programs more understandable, due to explicitly expressed design decisions, and less sensitive to requirement changes.

There are some areas of future work. First, the fine-grained scale between object composition and inheritance renders the common visibility modifiers **public** and **protected** too coarse, so that a more sophisticated visibility concept is desirable. Such a refined visibility concept may also solve encapsulation problems as described in [SM95]. Another interesting area is to investigate the space of possible composition property combinations for invalid combinations, which need to be rejected at compile-time. Finally, an interesting extension of the CR concept would be to also allow CRs to dictionary entries, so that the dictionary keys take the roles of field names,

and the corresponding values the role of field values.

CHAPTER 3

Hierarchical Decomposition: Collaborating Entities

This chapter shares material with the paper ‘Delegation Layers’ [Ost02] which has been presented at ECOOP 2002.

In this chapter we will enhance the common mechanism for hierarchical decomposition with respect to a kind of ‘higher-order’ decomposition, that is, we will provide means to organize and decompose the decompositions themselves. This is very useful in order to reason about sets of collaborating modules because with these mechanisms collaborating modules have an explicit representation in the programming language. In more detail, we will generalize the object-oriented notions of subtyping, subsumption, polymorphism and delegation to *sets* of collaborating classes. Technically, this is realized by combining class-based delegation with the idea of virtual classes. Thereby, it is possible to organize collaborating classes in layers that can be combined via delegation, hence the approach is called *delegation layers*.

3.1 Introduction

In the early days of object-oriented programming there has been a general agreement that single classes should be the primary unit of organization and reuse. However, over the years it has been recognized that a slice of behavior affecting a set of collaborating classes is a better unit of organization than a single class. In the face of these insights, mainstream programming languages have been equipped with lightweight linguistic means to group sets of related classes, for example name spaces in C++ [ES95] or packages and nested classes in Java [AG96]. On the other hand, the research community has developed a great deal of models related to *collaboration-* or *role-model based design*, for example [BC89, HHG90, RG98, VN96].

Our point of view is that we should not try to invent a completely new kind of module for grouping classes just to realize (sooner or later) that we need means to express variants, hide details, have polymorphism etc. Instead, we propose a model within which all the concepts that proved so useful for *single* classes/objects, for example inheritance, delegation, late binding, and subtype polymorphism, automatically apply to *sets* of collaborating classes and objects.

In particular, we deal with the question of how sets of collaborating classes can be defined and composed in terms of different variants (*layers*) of a base collaboration. The running example in this chapter is a graph collaboration with classes like `Node` and `Edge` and variations of this collaboration for colored graphs and weighted graphs.

One of the most advanced approaches with respect to our goals is the *mixin layer* approach by Smaragdakis and Batory [SB98]. Mixin layers allow (a) sets of classes (which represent a particular collaboration and are implemented as nested classes of an outer class) to inherit from other sets of classes, and (b) the composition of different variants of a base collaboration. With regard to our running example, this means we can (a) implement a `Graph` collaboration with `Node` and `Edge` classes and refine the `Graph` collaboration to `ColoredGraph` or `WeightedGraph` via inheritance, and (b) combine `ColoredGraph` and `WeightedGraph` to a `ColoredWeightedGraph`.

Technically, the most important difference between our *delegation layer* approach and mixin layers is that the mixin layer notion of multi-class mixin-inheritance is replaced by multi-object delegation¹. This can be tentatively summarized as: With mixin layers, everything happens on classes at compile time, whereas with delegation layers, everything happens on objects at runtime.

This has a deep impact on the semantics and expressiveness of the model. In particular, delegation layers have the following two properties:

- **Polymorphic runtime composition:** In our approach, a collaboration is composed at runtime by combining different delegation layers. Since delegation layers are subject to subtype polymorphism, the code which combines the layers is decoupled from the specific layers to be composed. For example, we may combine a `ColoredGraph` with a `Graph g`, but at runtime, `g` may actually refer to an instance of `WeightedGraph`.
- **Local on-the-fly extensibility:** We can extend a group of collaborating objects' behavior on-the-fly, whereby these behavior extensions are local, meaning that after the extension both the original and modified behavior of the object group are simultaneously accessible. For

¹Please note that in contrast to the frequent use of the term delegation as a synonym for forwarding semantics, in this thesis it stands for dynamic, object-based inheritance as defined in [Lie86].

example, we may have an existing graph instance g with a set of node and edge objects and extend g with all its nodes and edges to be a colored graph cg . After the extension, the nodes and edges of the graph behave as a colored graph if they are accessed via cg and as a usual graph if they are accessed via g . We may even have multiple independent color extensions of a specific graph denoting different colorings of the same graph.

These properties are consequences of the runtime semantics of delegation layers. In addition, our approach eliminates two subtle flaws of the mixin layer approach related to polymorphism and consistency:

- **Polymorphism:** We define a notion of subtyping among collaborations which guarantees substitutability and allows us to use a compound collaboration where an instance of a particular layer is expected if and only if this layer is a part of the compound collaboration. For example, a graph that is both colored and weighted can be used where a colored graph is expected. Thus the advantages of standard OO subtyping (reusability, decoupling etc.) are transferred to collaboration inheritance. In general, this property does not apply for mixin layers.
- **Composition consistency:** Our approach guarantees that all operations inside a compound collaboration are applied to the composite collaboration rather than to a specific layer alone. In particular, this proposition holds for constructor calls, thereby eliminating a composition anomaly of mixin layers.

Composing and extending collaborations at runtime yields type safety and consistency questions not emerging with compile time composition. In order to give answers to these questions, our model combines delegation techniques with virtual classes [MMP89], family polymorphism [Ern01], and a wrapper technique that is based on the idea of lifting and lowering as described in [MSL01]. Although - to the best knowledge of the author - delegation has never been combined with virtual classes before, the interplay between these two mechanisms is elegant and natural.

The rest of this chapter is structured as follows: Sec. 3.2 elucidates the concept of composable collaborations and gives a short overview of mixin layers. In addition, it emphasizes the weaknesses of mixin layers with respect to the aforementioned benefits we aim at. Sec. 3.3 and 3.4 introduce simple variants of delegation and virtual classes with family polymorphism as extensions of Java [AG96]. Sec. 3.5 shows how delegation and virtual classes interact and introduces the notion of delegation layers. Sec. 3.6 elaborates on on-the-fly extensions and the impact of delegation layers on sharing and aliasing. Sec. 3.7 discusses related work. Sec. 3.8 summarizes and indicates areas of future work.

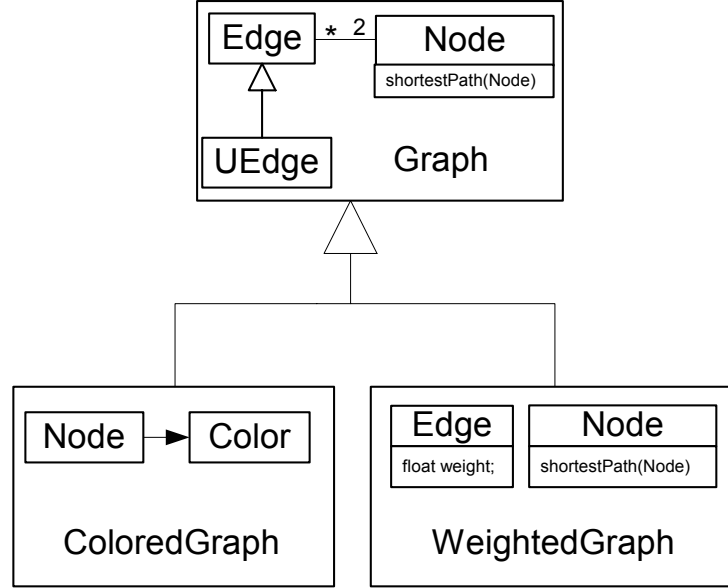


Figure 3.1: Collaboration inheritance

3.2 Collaboration Composition and Mixin Layers

The rationale behind collaboration composition is that sets of collaborating classes can be defined and composed in terms of different variants (*layers*) of a base collaboration. Consider the situation in Fig. 3.1. It shows two collaborations **ColoredGraph** and **WeightedGraph** that inherit from a base collaboration **Graph**. The **Graph** collaboration defines classes **Node**, **Edge** and **UEdge**. The graphs in this example are assumed to be directed in general, and the class **UEdge** represents undirected edges which enter themselves in the adjacency list of *both* nodes. The subcollaborations **ColoredGraph** and **WeightedGraph** extend the base collaborations by defining classes that extend (i.e., are subclasses of) the base collaborations' classes. For example, the class **ColoredGraph.Node** extends **Graph.Node** by an additional association to **Color**. The class **WeightedGraph.Edge** adds a field `float weight` and **WeightedGraph.Node** overrides the inherited `shortestPath()` method in order to consider the edge weights.

The key issue is the ability to compose different variants of a base collaboration. For example, we may want to create a graph that is both colored and weighted by means of the collaborations in Fig. 3.1. Fig. 3.2 demonstrates the desired semantics of `WeightedGraph(ColoredGraph(Graph))`: The collaborations are organized in layers according to the order in the composition expression; i.e. the outermost **WeightedGraph** collaboration is at the bottom, in the middle the **ColoredGraph**, and the **Graph** collaboration


```
class Graph {
public:
    class Node {
    public:
        NodeList shortestPath(Node *t) {...}
    };
    class Edge { ... };
    class UEdge: public Edge { ... };
};
template <class SuperGraph>
class ColoredGraph : public SuperGraph {
public:
    class Node: public SuperGraph::Node {
        Color color;
        ...
    }
};
template <class SuperGraph>
class WeightedGraph : public SuperGraph {
    class Edge: public SuperGraph::Edge {
        float weight;
        ...
    }
    class Node: public SuperGraph::Node {
    public:
        NodeList shortestPath(Node *t) {...}
    }
};
typedef ColoredGraph<WeightedGraph<Graph> > CWG;
typedef ColoredGraph<Graph> CG;
```

Figure 3.3: Graph Example with C++ mixin layers

```
class Client {
    void createTransitiveHull(Graph *g, Graph::Node *n) {
        ... Graph::Node *m = currentNode->neighbor(i); ...
        if ( ! n->isNeighbor(m) ) {
            Graph::Edge *e = new Graph::Edge(n,m); ...
        }
    }
};
```

Figure 3.4: Restricted polymorphism in the mixin layer approach

- **Polymorphism:** Mixin layers have two flaws concerning polymorphism. First, subtyping among collaborations is too restrictive. Consider for example the two types `CWG` and `CG` in Fig. 3.3. Although a colored weighted graph of type `CWG` has all features of a colored graph type (`CG`), the former one is no subtype of the latter one². Secondly, in the cases where subtyping is possible, the effect of substitution is not as expected. Consider the example in Fig. 3.4. If the method `createTransitiveHull` is called with an instance of a colored graph `CG`, the `new` statements will still create instances of `Graph.Node` and `Graph.Edge` rather than of their corresponding implementations for colored graphs. The problem is that the constructors in a `new` statement are statically bound to a specific implementation and we have no means to express that the `new` statements should instantiate the classes that are appropriate in the specific collaboration represented by `g`. Please note that the factory pattern [GHJV95] is in general no satisfactory solution because this pattern needs to be anticipated and cannot be applied to superclasses (i.e., we cannot retrieve the superclass of a nested class via a factory object).
- **Composition consistency:** Inside a compound collaboration, all operations should be applied to the composite collaboration rather than to a specific layer. This is in general not true for constructor calls in mixin layers. Consider for example the compound `UEdge` class in Fig. 3.2. In a weighted graph, the weight property should of course also apply to `UEdge`. This means that in the context of a weighted graph, `UEdge` should inherit from the *compound* `Edge` rather than from `Graph.Edge` (see also Fig. 3.2). The same argument also applies to `new` statements inside a collaboration. If the class to be created is a participant of the collaboration, we expect a corresponding `new` statement to create an instance of the respective *compound* participant class. However, in the mixin layer approach the constructor calls refer to a fixed implementation in both cases. For example, the `UEdge` class in Fig. 3.3 is always a subclass of `Graph.Edge` and not of `WeightedGraph.Edge`, even in the context of the weighted graph collaboration.

The second problem has also been acknowledged in [SB98]. We think that it can be seen as a variant of the *self problem* [Lie86], a.k.a. *broken delegation* [HOT97]: In a composite component, all actions should be applied to the composite component, rather than to an individual part of it. The original formulation of the self problem refers to method calls; in our case, it refers to constructor calls.

²However, this flaw can be attributed to the C++ template implementation and is no conceptual weakness of mixin layers

3.3 Delegation

This section introduces the first building block of delegation layers, namely a simple variant of delegation as an extension of Java. Delegation means that objects inherit from other objects, with roughly the same semantics as classes which inherit from other classes. An object *o* that inherits from (*delegates to*) another object *p* is called *child* of *p*, and *p* is a *parent* of *o*.

To make the discussion simple, we restrict ourselves to *static delegation*, meaning that the parent of an object can be set at runtime, but once the parent reference is initialized, it cannot be changed, similar to a **final** variable in Java. This restriction avoids many problems which are not in the scope of this model; see Sec. 2.5 of the previous chapter.

Consider the situation in Fig. 3.5. It shows classes **Graph**, **ColoredGraph** and **WeightedGraph** as well as some demonstration code that uses delegation. In our approach, we unify standard inheritance and delegation as follows: In a **new WeightedGraph()** expression for a class **WeightedGraph** as in Fig. 3.5, we may *optionally* specify a parent object (delimited by **<>**) that has to be a subtype of the original superclass **Graph**. For example, let **ColoredGraph** be another subclass of **Graph**. Then **new WeightedGraph()** creates an instance of **WeightedGraph** with superclass **Graph** (usual semantics), and **new WeightedGraph<cg>()** creates an instance of **WeightedGraph** with parent **cg** (see Fig. 3.5). In the latter case, the parent object replaces the superclass.

The unification of delegation and inheritance has two advantages. First, the usage of a class with a different superclass than initially intended does not have to be anticipated. Second, we have a *default* superclass/parent, so that it becomes easier to create instances of such a class. In the remainder of this chapter, we will treat the direct instantiation of an object (without specifying a parent object) as an abbreviation for assigning an instance of the superclass as parent object. For example, **new ColoredGraph()** is an abbreviation for **new ColoredGraph<new Graph>()**³. The parent object of an object *o* is always available via the implicit **super** field *o.super*.

The key issue in combining classes and objects via delegation is the treatment of the **this** and **super** pseudo variables. This is illustrated in the demonstration code in Fig. 3.5 and in Fig. 3.6: The **this** pseudo variable refers to the *receiver* of a method call, and **super** refers to the (possibly dynamically assigned) parent/superclass. The implications are illustrated by the **printInfo()** calls in Fig. 3.5. It is important to understand that the value of **this** is not fixed but depends on the receiver of a message. For example, in the context of a method call to **cg**, all **this** pointers refer to the instance of **ColoredGraph** rather than to the **WeightedGraph** instance.

³For the sake of simplicity we assume that every class has only a single no-argument constructor.

```
class Graph {
    private String info = "SomeInfo";
    public String setInfo(String s) { info = s; }
    String toString() { return "Graph, info="+info; }
    void printInfo() { print(this.toString()); }
}
class ColoredGraph extends Graph {
    String toString() { return "Colored"+super.toString();}
}
class WeightedGraph extends Graph {
    String toString() { return "Weighted"+super.toString();}
}
// demo code
Graph g = new WeightedGraph();
g.printInfo(); // prints "WeightedGraph, info=SomeInfo"
Graph cg = new ColoredGraph();
cg.printInfo(); // prints "ColoredGraph, info=SomeInfo"
Graph wg = new WeightedGraph<cg>();
wg.printInfo(); // prints "WeightedColoredGraph, info=SomeInfo"
cg.setInfo("OtherInfo");
wg.printInfo(); // prints "WeightedColoredGraph, info=OtherInfo"
ColoredGraph cg2 =
    (ColoredGraph) wg; // succeeding cast due to transparency
```

Figure 3.5: Code example for delegation

Delegation is more than just composing classes at runtime. An important property of delegation is that parent objects may be *shared*. In Fig. 3.5, both the `cg` instance variable and the parent reference of `wg` refer to the same object. This is demonstrated by the `cg.setInfo()` call which affects `wg` due to the shared `ColoredGraph` object.

A property of delegation that has been postulated by Büchi and Weck [BW00] is *transparency*, meaning that an object is a subtype of the *dynamic* type of its parent. This property is relatively straightforward in the context of static delegation⁴. In our example (Fig. 3.5), transparency means that the dynamic cast in the last line succeeds. We will see that the incorporation of transparency supports the elimination of the polymorphism problem as indicated in Sec. 3.2 (“better support for polymorphism”).

For further details about the integration of delegation into a statically typed language, we refer to the existing approaches, e.g., [Kni99, BW00] as well as Chap. 2 of this thesis.

⁴In models that support full dynamic delegation, transparency implies serious typing problems, see Sec. 2.5.

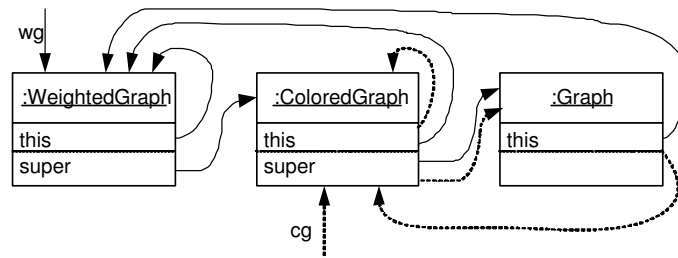


Figure 3.6: Meaning of **this** and **super** in a delegation relationship as in Fig. 3.5. The non-dashed lines represent the behavior if the objects are accessed via **wg** and the dashed lines represent the behavior if the objects are accessed via **cg**.

3.4 Virtual Classes

Virtual classes are the second important building block for delegation layers. Virtual classes are a concept from the Beta programming language [MMP89, MMPN93] (in Beta known as *virtual pattern*). The basic idea is that the notions of overriding and late binding should also apply to nested classes, similarly to overriding and late binding of methods.

Consider the class **Graph** in Fig. 3.7. The nested classes **Node**, **Edge** and **UEdge** are declared as *virtual classes* with a **virtual** modifier (corresponds to **<** in Beta), meaning that these classes can be overridden by subclasses of the enclosing class. In contrast to methods in Java, nested classes are not virtual by default, because a virtual class has important typing implications.

The classes **ColoredGraph** and **WeightedGraph** in Fig. 3.7 override virtual classes of their superclass. The meaning of an **override** declaration (corresponds to **::<** in Beta) such as the **override class Node** declaration in **ColoredGraph** is that - in the context of **ColoredGraph** - the class **Graph.Node** is replaced by the class **ColoredGraph.Node**. The latter one is automatically a subclass of the former one, and hence has all methods and fields of **Graph.Node** plus an additional color field. Since an overriding virtual class is automatically a subclass of the overridden class, the single inheritance link is already allocated, such that an overriding class cannot extend another class.

As mentioned before, the rationale behind virtual classes is that overriding and late binding should uniformly apply to methods as well as virtual classes. Late binding of *methods* means that the receiver *object* determines the method implementation to be executed. If this principle is applied to virtual classes, it becomes clear that virtual classes should also be properties of *objects* of the enclosing class, rather than properties of the enclosing class itself. This is in the vein of the family polymorphism approach [Ern01].

```
class Graph {
    virtual class Node {
        void foo() { Edge e = new Edge(); }
        NodeList shortestPath(Node n) { ... }
    }
    virtual class Edge { ... }
    virtual class UEdge extends Edge { ... }
}

class ColoredGraph extends Graph {
    override class Node { Color color; ... }
}

class WeightedGraph extends Graph {
    override class Node {
        NodeList shortestPath(Node n) {...}
    }
    override class Edge { float weight; ... }
}
```

Figure 3.7: Virtual classes

Virtual classes being properties of an object means that all references to a virtual class are resolved via an instance of the enclosing class. In our approach, we apply the Java scoping rules for method calls to virtual classes as well, meaning that all references to a virtual class are implicitly resolved via the corresponding `this`. Fig. 3.8 makes the implicit scoping of Fig. 3.7 explicit. For example, the type declaration `Edge e` in `foo()` is a shorthand for `Graph.this.Edge` (the notation `Graph.this` refers to the instance of the enclosing class). Similarly, `UEdge` in `Graph` is a subclass of `this.Edge` (rather than of `Graph.Edge`) and `ColoredGraph.Node` extends `super.Node` (rather than `Graph.Node`).

Consequently, `Graph.Node` is no longer a valid type annotation: The treatment of virtual classes as properties of objects stretches out to the client code of a class as well. For example, the family polymorphism version of Fig. 3.4 is shown in Fig. 3.9: Type annotations and constructor calls for virtual classes are all redirected via an instance of the enclosing class.

For type checking reasons, variables that are used inside type declarations have to be `final`. Otherwise, the type `g.Node` of a variable could change due to an update of `g`.

In contrast to Fig. 3.4, `createTransitiveHull` in Fig. 3.9 works with arbitrary subclasses of `graph` and without compromising type safety. Similarly, a statement like `node1.shortestPath(node2)` can be statically proved type-safe, if `node1` and `node2` are both of type `g.Node`, although at run-time `g` may refer to an instance of an arbitrary *subclass* of `Graph`. This

```

class Graph {
    virtual class Node {
        void foo() { Graph.this.Edge e = new Graph.this.Edge(); }
        NodeList shortestPath(Graph.this.Node n) { ... }
    }
    virtual class Edge { ... }
    virtual class UEdge extends this.Edge { ...}
}

class ColoredGraph extends Graph {
    override class Node extends super.Node { Color color; ... }
}

class WeightedGraph extends Graph {
    override class Node extends super.Node {
        NodeList shortestPath(WeightedGraph.this.Node n) {...}
    }
    override class Edge extends super.Edge { float weight; ... }
}

```

Figure 3.8: Virtual classes are properties of *objects* of the enclosing class.

```

class Client {
    void createTransitiveHull(final Graph graph, graph.Node n) {
        ... graph.Node m = currentNode.neighbor(i); ...
        if ( ! n.isNeighbor(m)) {
            graph.Edge e = new graph.Edge(n,m); ...
        }
    }
}

```

Figure 3.9: Family polymorphism version of Fig. 3.4

demonstrates that the treatment of virtual classes as properties of objects is an alternative to other approaches for retaining type safety in the presence of virtual classes, such as final bindings [Tor98] or type-exact variables [BOW98].

More details about virtual classes, family polymorphism, and their typing implications can be found in [Ern01] and [OCRZ03].

3.5 Delegation Layers

Delegation layers are the result of combining delegation with virtual classes. In the following, we want to elaborate on the interplay between these two mechanisms.


```
main() {  
    Graph cg = new ColoredGraph();  
    final Graph g = new WeightedGraph<cg>();  
    g.Node n = ...;  
    new Client().createTransitiveHull(g,n);  
    ColoredGraph cg2 =  
        (ColoredGraph) g; // succeeding dynamic cast  
}
```

Figure 3.10: Delegation layers

Reconsider the semantics of our virtual class mechanism as demonstrated in Fig. 3.7 and 3.8. All references to virtual classes are actually resolved via the implicit **this** and **super** pseudo variables of the enclosing object.

Our delegation mechanism implies that the denotation of **this** and **super** can be altered at runtime. Consider the second **new** expression in Fig. 3.10. It creates an instance of **WeightedGraph** and assigns an instance of **ColoredGraph** as parent of the weighted graph object. The meaning of **this** and **super** in the context of **g** has already been illustrated in Fig. 3.6.

The crucial point is that virtual classes and delegation, if combined, interact due to their influence/dependency on the semantics of **this** and **super**. The semantics of this interaction, which is illustrated in Fig. 3.11, can be derived from Fig. 3.6 and 3.8. Consider, for example, the superclass declaration “**extends super.Node**” in **WeightedGraph.Node** (Fig. 3.8). In the context of **g**, **super** refers to an instance of **ColoredGraph**, therefore the parent of **WeightedGraph.Node** is an instance of **ColoredGraph.Node**. Similarly, the superclass declaration “**extends this.Edge**” in **Graph.UEdge** binds the parent of **UEdge** to an instance of **WeightedGraph.Edge**. The boxes with bold frame in Fig. 3.11 represent the composite classes **g.Node**, **g.Edge**, and **g.UEdge**.

Generally speaking, the combination of virtual classes and delegation effects the delegation relationship to spread to the nested virtual classes of the enclosing object. This is exactly the semantics that is required to obtain the composition behavior indicated in Fig. 3.2; cf. Fig. 3.2 and Fig. 3.11.

Let us revisit the delegation layer approach with respect to the goals stated in the introduction (except on-the-fly extensibility, which is the subject of the next section).

- **Polymorphic runtime composition:** Due to the use of delegation, the composition happens at runtime, and the composition code does not need to know the exact classes of the layers. Note, for example, that the static type of the parent reference **cg** in Fig. 3.10 is **Graph**, although it actually refers to an instance of **ColoredGraph**, which might have been passed as a method argument as well.

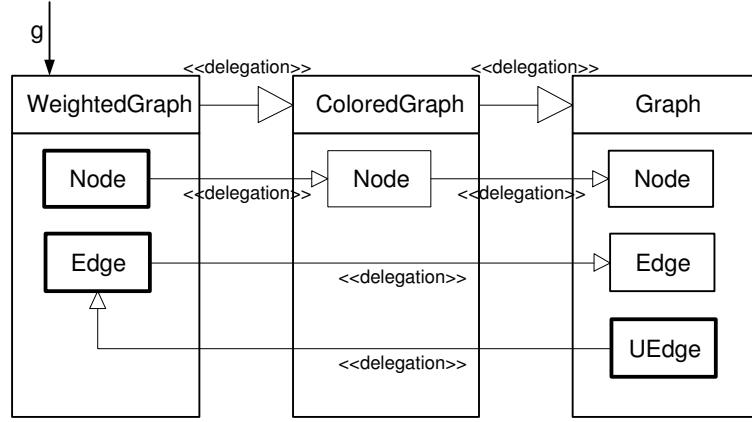


Figure 3.11: Recursive delegation

- **Polymorphism:** In Sec. 3.2 two different shortcomings related to polymorphism with mixin layers have been. The first one is the restricted subtyping. For example, a colored weighted graph **CWG** is no subtype of a colored graph **CG** in Fig. 3.3. Compare this with the last line in Fig. 3.10. Due to the introduction of transparency (see Sec. 3.3), an instance of a compound collaboration can be dynamically casted to the type of all participants. In general, if $C = C_1(C_2(\dots(C_n)\dots))$ is a composed collaboration, C is a subtype of C_i for $i = 1, \dots, n$ in the delegation layer approach. With mixin layers, on the other hand, C is only a subtype of $C_i(C_{i+1}(\dots(C_n)\dots))$ for $i = 1, \dots, n$.

The second flaw is that the effect of subsumption is not as expected, exemplified by Fig. 3.4. This problem ceases to exist in our approach, because the class which is instantiated in response to a constructor call is determined at runtime, depending on the instance of the enclosing class (see Fig. 3.9).

- **Composition consistency:** We postulated that inside a compound collaboration, all operations should be applied to the composite collaboration rather than to a specific layer. With mixin layers, this property is violated, cf. the discussion in Sec. 3.2. With delegation layers, the class **UEdge** is a subclass of the *compound Edge* class (see Fig. 3.11), and the `new Edge()` statement in `foo()` creates an instance of the *compound Edge* (cf. Fig. 3.7 and 3.8).

3.6 Hot State and On-the-fly Extensions

So far, we have avoided to introduce state into the classes in our examples. At first sight, it seems as if this implies both a semantic and a typing

```
class Graph {
    Node n;
    Node getNode() { return n; }
    void setNode(Node n) { this.n = n; }
    ... // as in Fig. 7
}
// demo code
final Graph g = new Graph();
g.setNode(new g.Node());
g.Node node    = g.getNode(); // OK
final ColoredGraph cg = new ColoredGraph<g>();
cg.Node cnode  = cg.getNode(); // Type Error ?
cnode.color    = Color.RED;    // ??
```

Figure 3.12: Potential problems due to hot state

problem. Consider the code in Fig. 3.12. The `Node n` instance variable is initialized to an instance of `Graph.Node` (and not `ColoredGraph.Node`). However, if this graph instance was extended by an instance of `ColoredGraph`, the same (identical) node would suddenly have to be a colored node.

The cause of this problem is that the type of the instance variable `Node n` is non-constant. Recall that `Node n` is an abbreviation for `this.Node n` (see Sec. 3.4). Hence `this` refers to an instance of `Graph` if `n` is accessed via `g`, and refers to an instance of `ColoredGraph` if `n` is accessed via `cg`. We call such state whose type depends on the enclosing `this` *hot state*.

We found a mechanism that turns this problem into a feature. It is based on the idea of *lifting* and *lowering* as described in [MSL01] but adapted to the specific needs of our model. The basic idea is that, in the context of a colored graph, a node `n` can be automatically lifted to a colored node by creating an instance of `ColoredGraph.Node` that delegates to `n`. In order to make this approach sound, it is essential that two subsequent liftings for the same node yield the same colored node.

Fig. 3.13 shows pseudo code indicating the operational semantics of the lifting and lowering operation. Please note that the programmer does not write this code: The code is just an illustration of the language semantics in terms of OO constructs.

Every class `C` maintains a map (hashtable) for every virtual class which is overridden in that class. For example, `ColoredGraph` has a map `nodeMap`, and `WeightedGraph` has the maps `nodeMap` and `edgeMap`. In addition, a class `C` has a lifting and a lowering operation `liftV(V v)` resp. `lowerV(V v)` for each virtual class `V` that is (a) defined or (b) overridden in `C`. In case (a), the lifting and lowering operations simply return their argument. In case (b), the lifting operation lifts an instance of the base virtual class (e.g., `Graph.Node`)

```
class Graph {
    // begin internal structure pseudocode
    Graph.Node liftNode(Graph.Node n) { return n; }
    Graph.Node lowerNode(Graph.Node n) { return n; }
    Graph.Edge liftEdge(Graph.Edge e) { return e; }
    Graph.Edge lowerEdge(Graph.Edge e) { return e; }
    // end internal structure pseudocode
}
class ColoredGraph extends Graph {
    // begin internal structure pseudocode
    private Map nodeMap = new HashMap();

    ColoredGraph.Node liftNode(Graph.Node n) {
        ColoredGraph.Node result =
            (ColoredGraph.Node) nodeMap.get(n);
        if (result == null) {
            result =
                new ColoredGraph.Node<super.liftNode(n)>();
            nodeMap.put(n,result);
        }
        return result;
    }
    Graph.Node lowerNode(ColoredGraph.Node n) {
        Graph.Node result = super.lowerNode(n.super);
        nodeMap.put(result, n);
        return result;
    }
    // end internal structure pseudocode
}
```

Figure 3.13: Lifting details

to an instance of the overriding virtual class (`ColoredGraph.Node`), and the lowering operation lowers an instance of the overriding virtual class (e.g., `ColoredGraph.Node`) to an instance of the base virtual class (`Graph.Node`).

The semantics of the lifting and lowering operations is indicated in Fig. 3.13: The lifting and lowering operations in `Graph` simply return their argument. The more interesting case are the lifting and lowering operations in `ColoredGraph`, which override their corresponding implementations in `Graph`.

In `liftNode()`, a lookup in the map determines whether the same node has ever been lifted before. In this case, the corresponding instance of `ColoredGraph.Node` is directly returned. Otherwise, an instance of `ColoredGraph` that delegates to the node instance is created, stored in the map, and returned. The lookup in the map ensures that subsequent liftings for the same node yield the same `ColoredNode` wrapper.

The `lowerNode()` operation is the counterpart of `liftNode()`. It stores the `ColoredGraph` part of the node in the map and recursively asks its parent `super` to lower the parent of the node (`n.super`).

In the `liftNode()` operation, the parent object of the wrapper object is `super.liftNode(n)` rather than `n`, and in `lowerNode()`, the method returns `super.lowerNode(n.super)` rather than `n.super`. This ensures the mechanism will work when a class `C` overrides a virtual class that has already been overridden in the superclass of `C`. The anchor of the recursions are the `liftNode()` and `lowerNode` operations in the class that introduces the virtual class (in this case `Graph`, which simply return their argument, see Fig. 3.13).

What are the appropriate places to apply lifting and lowering operations? We think that the only reasonable solution is to apply it whenever hot state is evaluated; that is, the r-value of a hot instance variable in an expression `node` is actually `liftNode(node)`, and the l-value of a hot variable in an assignment `node = anExpression` is actually `node = lowerNode(anExpression)`. In our example, this means that the implementation of `getNode()` returns `this.liftNode(n)` rather than `n`, and the implementation of `setNode()` assigns the result of `lowerNode(n)` to `this.n`.

The calls to `liftNode()` and `lowerNode` are subject to late binding, because the respective implementations of `ColoredGraph` overrides the implementations in `Graph`. For example, the call `g.getNode()` in Fig. 3.12 yields a call to `Graph.liftNode()`, while the call `cg.getNode()` yields a call to `ColoredGraph.liftNode()`.

An important invariant of lifting and lowering is that the function combination `lowerV(liftV(v))` is the identity function, such that a statement like `node = node`, which translates to `node = lowerNode(liftNode(node))`, has the expected meaning.

This approach preserves static type safety because the lifting operation ensures that the evaluation of a hot instance variable yields an instance of the type which is appropriate for the respective context by dynamically creating and maintaining wrappers that delegate to the base objects. The hash table guarantees that we do not lose the state and identity of the individual parts of a delegation chain. Finally, the lowering operation guarantees consistency in the sense that all objects will only interact with other objects from the same family. For example, if we would execute the statement `g.setNode(new cg.Node())` with `g` and `cg` as in Fig. 3.12, and we would *not* apply lowering, we would suddenly have a colored node in a context `g` that does not assume color properties. A subsequent call like `g.getNode().setNeighbor(new g.Node())` would expose the inconsistency because the original colored node would assume that its neighbor nodes would also be colored nodes.

However, this approach is much more than a fix to preserve type safety. Let us look at a more interesting example. Consider the code in Fig. 3.14.

```
class Graph {
    Node[] nodes;
    void setNode(Node n, int i) {
        nodes[i] = n; // effectively assigns this.lowerNode(n)
    }
    Node getNode(int i) {
        // effectively returns this.liftNode(nodes[i])
        return nodes[i];
    }

    virtual class Node {
        Edge[] edges;
        Edge getEdge(int i) {
            // effectively returns this.liftEdge(edges[i])
            return edges[i];
        }
    }
    virtual class Edge {
        Node n1, n2;
        Node getTargetNode() {
            // effectively returns this.liftNode(n2)
            return n2;
        }
    }
}
class ColoredGraph extends Graph ... // as in Fig. 7
```

Figure 3.14: A graph with hot state

It shows a graph class which stores a graph as a list of nodes. A node has a list of incident edges and an edge stores its source and target node. The comments in the code indicate the places where the lifting and lowering actually takes place.

Let us suppose we want to determine the chromatic number⁵ and/or a corresponding coloring for a specific graph. Let us further suppose we have an appropriate algorithm in a class **GraphColoring** as indicated in Fig. 3.15. Of course, the algorithm is directly applicable to any graph which has been instantiated as **ColoredGraph**. However, let us suppose that this is not the case for our sample graph because, say, we just want to know the chromatic number and are not interested in the coloring itself and do not want to waste the corresponding memory. Another reason might be that we want a graph that has different independent colorings with different meanings.

The demo code in Fig. 3.15 shows how an arbitrary graph can be ex-

⁵The minimum number of colors needed to color the vertices of a graph such that no two adjacent vertices have the same color.

```
class GraphColoring {
    int chromaticNumber(final ColoredGraph g) {
        ...
        g.Node node = g.getNode(i);
        node.color = Color.RED; // statically safe
        ...
    }
    void randomColoring(final ColoredGraph g) {
        ...
    }
}
// demo code
Graph g = ...;
GraphColoring coloring = new GraphColoring();
ColoredGraph cg1 = new ColoredGraph<g>();
int i = coloring.chromaticNumber(cg1);
...
ColoredGraph cg2 = new ColoredGraph<g>();
coloring.randomColoring(cg2);
```

Figure 3.15: Independent on-the-fly extensions of a graph

tended on-the-fly with the mechanisms of our approach. The color extension is only visible via `cg1` and `cg2`, respectively. The state and behavior of the graph remains unchanged if it is accessed via `g`. Please note how easy it is to create two completely independent colorings (chromatic and random coloring) for a specific graph instance. Due to subtype polymorphism, these extensions are also decoupled from the specific graph instance in the sense that `g` may also refer to an instance of `WeightedGraph` or even `ColoredGraph`. In the latter case, the extension would yield a coloring which is independent from the original coloring of `g`.

The last example in this section does not introduce new features but emphasizes two important properties of our approach: The ability to extend a collaboration which has already been extended (orthogonality), and transparent simultaneous behavior extensions for all objects of a collaboration instance.

Suppose we want to observe the progress of the coloring algorithm on the screen in case the respective graph is currently displayed. In other words, we want to be notified whenever the `setColor()` method is invoked for a node of that graph.

Consider the code in Fig. 3.16. It introduces an appropriate interface `ColorObserver` that is implemented by `GraphDisplay`. The class `NotifyingGraph` extends the behavior of all color nodes such that the `ColorObserver` is notified whenever the color of that node is changed. The demonstration code creates a `Graph g` and extends `g` to be a colored graph in the

```

interface ColorObserver {
    void colorChanged(final ColoredGraph cg,
                      cg.Node node, Color color);
}
class GraphDisplay implements ColorObserver { ... }
class NotifyingGraph extends ColoredGraph {
    ColorObserver o;
    public NotifyingGraph(ColorObserver o) { this.o = o; }

    override class Node {
        void setColor(Color color) {
            super.setColor(color);
            o.colorChanged(NotifyingGraph.this, this, color);
        }
    }
}
// demo code
Graph g = ...; GraphDisplay display = ...;
GraphColoring coloring = new GraphColoring();
ColoredGraph cg1 = new ColoredGraph<g>();
if (screenDisplay) cg1 = new NotifyingGraph<cg1>(display);
int i = coloring.chromaticNumber(cg1);

```

Figure 3.16: Adding notifier functionality

context of `cg1`. The colored graph `cg1` is again extended with the notifier behavior if the variable `screenDisplay` evaluates to `true`.

Please note that the graph `cg1` that is potentially extended with the `NotifyingGraph` functionality is already an extended version of the original graph instance `g`.

The type of `cg1` is `ColoredGraph` and not `NotifyingGraph`. Nevertheless, the extensions defined by `NotifyingGraph` spread through all further actions via `cg1`. In particular, all `setColor()` invocations in the coloring algorithm are dispatched to the `setColor()` redefinition in `NotifyingGraph`, although the author of the coloring algorithm does not know anything about the existence of `NotifyingGraph`.

The powerful expressiveness of on-the-fly extensions is due to the fact that delegation layers allow simultaneous behavior extensions for *sets* of objects. To the best knowledge of the author, delegation layers are the first approach that enables such kind of operations.

3.7 Related Work

The relation to mixin layers [SB98], virtual classes [MMP89], family polymorphism [Ern01], and delegation [Lie86, Kni99, BW00, OM01] has already

been discussed in Sec. 1–4.

Java Layers [BCML02, CL01] are a Java-based implementation of mixin layers. Java Layers extend Java by supporting constrained parametric polymorphism and mixins. The authors acknowledge the composition consistency problem and propose different solutions (called *sibling pattern*), including a limited variant of virtual types and a naming convention approach, to cope with this problem. An interesting approach in Java Layers, which might also be useful for delegation layers, is their notion of *deep conformance*, which extends Java’s concept of interfaces to include nested interfaces.

Jiazzi [MFH01] is a system that does also allow classes to be composed in a mixin layer style at compile time. Jiazzi is especially related to our work because it addresses both the composition consistency and the polymorphism problem (see Sec. 3.2). Their proposal for the composition consistency problem is based on the *open class pattern*, a kind of design pattern that mimicks the constructor semantics of virtual types. An application that uses a particular layer (*package* in the terminology of [MFH01]) can be parameterized with different variants of this layer, thereby eliminating the polymorphism flaw of the original mixin layer idea. This is similar to the idea of parameterizing a method with a family object, as shown in Fig. 3.9. However, in contrast to delegation layers, composition and polymorphism in Jiazzi are pure compile-time / link-time concepts, there is no notion of subtyping polymorphism and subsumption among different variants of a layer.

In comparison with delegation layers, a practical advantage of all aforementioned compile-time approaches [SB98, BCML02, CL01, MFH01] is that it is very much easier to create an efficient implementation with little or no runtime overhead.

In general, *virtual classes* are an interesting alternative or complement to parametric polymorphism. Please note that this is not the main focus of our approach, in contrast to the approaches in [Tho97] and [BOW98]. Therefore we do not introduce additional language means to express virtual classes defined outside the enclosing class, e.g., virtual classes like `StackItem` that are later overridden with `String` or `Point` in order to create a stack of strings or a stack of points.

Pluggable composite adapters (PCA) [MSL01] are a language construct for on-the-fly adaptation of frameworks. A set of base objects can be dynamically extended with the functionality provided by a particular framework. The relations between base objects and framework objects are maintained by a lifting technique that is similar to the one proposed in this chapter. However, in PCA, objects are lifted to types that are in general unrelated to their original type, whereas with delegation layers, objects are lifted to subtypes that delegate to the original object. In contrast to delegation layers, it is not possible to *change* the behavior of the lifted objects.

Delegation layers can also be seen as a form of *aspect-oriented programming* [KLM⁺97]. A delegation layer defines functionality that affects the

behavior of a set of different classes and can thus be seen as a module for crosscutting concerns. In comparison with AOP languages like AspectJ [Asp03], delegation layers have a very limited joinpoint model. On the other hand, delegation layers are much more dynamic than other AOP languages. For example, in AspectJ it would also be possible to extend the `Graph` class with color functionality. However, in this case *all* graphs would automatically be colored graphs; it would not be possible offhand to access a graph simultaneously both as a graph and a colored graph, or create independent colorings as in Fig. 3.15. The same argument applies to the notification extension in Fig. 3.16. In AspectJ, the notification would automatically apply to *all* graphs and it would require additional measures (e.g., conditional statements in the form of `if (notifyEnabled) ...`) to be able to choose at runtime which graphs feature the notification behavior.

A number of approaches focus on the evolution of single objects or single classes. The basic idea of the *context relationship* [SPL98] is that if a class `C` is context-related to a base class `B`, then `B`-objects can get their functionality dynamically altered by `C`-objects. A `C`-object may be explicitly attached to a `B`-object, or it may be implicitly attached to a group of `B`-objects for the duration of a method invocation. In *Rondo* [Mez97], the behavior of single objects can be altered at runtime by means of so-called *adjustments*. With *predicate classes* [Cha93], an object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. If an object is modified, the classification of an object can change, yielding in a different behavior of the object.

There have been a number of proposals related to *collaboration-* or *role-based design* [Ree95, BC89, HHG90, RG98, Hol92, ML98]. In contrast to these approaches, delegation layers focus on the definition and on-the-fly runtime combination of collaboration *variants*.

3.8 Chapter Summary

In this chapter we proposed delegation layers, a new mechanism to define and combine sets of collaborating classes and objects. Since the modules to group such sets are classes and objects themselves, the concepts that proved so useful for single classes and objects - inheritance, delegation, late binding, instantiation, subtype polymorphism etc. - apply to sets of collaborating classes and objects as well.

Due to their strong runtime semantics, delegation layers are extremely flexible. In particular, the ability for local on-the-fly extensions, with which we can change the behavior of a *set* of objects (instead of a single object with classical delegation) seems to be very promising. We think that this is especially interesting with respect to the idea of aspect-oriented programming [KLM⁺97] because most AOP approaches are static in the sense that

there is no notion of applying aspects to individual runtime entities. In Chap. 4 and 5 we will build up on the lessons learned from delegation layers and equip them with additional means for aspect-oriented programming.

CHAPTER 4

Encoding Crosscutting Models

This chapter shares material with the paper ‘Integrating Independent Components with On-Demand Remodularization’ [MO02] which has been presented at OOPSLA 2002.

After the previous chapters of the thesis, which concentrated on improving hierarchical decomposition mechanisms, this chapter proposes language support to encode crosscutting models, that is, it proposes means to encode multiple independent hierarchies in a software system. This chapter is closely related to Chap. 5: Here we focus on encoding different models and means to mediate between these hierarchies. Chap. 5 builds on these results and adds means to specify more sophisticated interactions between these models in the form of so-called pointcuts and advices as well as support for a polymorphic composition of hierarchies by a new notion of aspect deployment.

The language described in these two chapters is called CAESAR, after Julius Caesar’s *divide et impera* policy.

4.1 Introduction

This chapter presents language support for separating and capturing generic application logic that is useful in several places within one application domain or even across application domains. Hence the attribute ‘generic’. In the terminology of standard component models such as CORBA, one would probably use terms such as vertical and horizontal facilities for the kind of generic application logic we target here.

Graph algorithms are a good match for the kind of the generic functionality we mean, since they are used in almost any application domain.

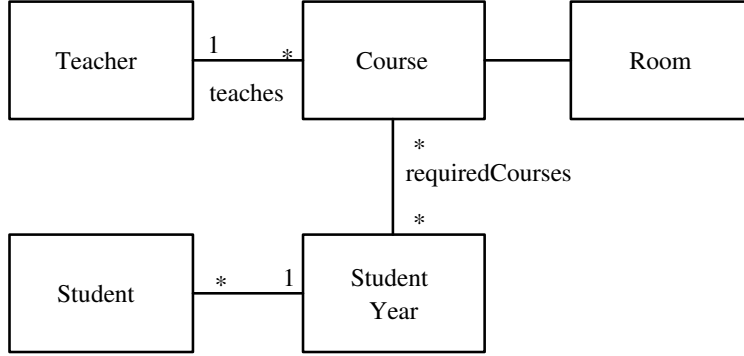


Figure 4.1: University Example

They are often even instantiated several times within the same application, whereby each instantiation might involve different units in the modular structure of the application in playing the roles of vertices and edges in the world of graph abstractions. For example, the university administration software in Fig. 4.1 can be viewed as a graph whose vertices are the courses and whose edges are gained by connecting any pair of courses that are taught by the same teacher or required by the same student year. This view on the module structure of the university software would be necessary on the demand of applying graph coloring¹ to compute an optimal assignment of time slots to courses. Other - eventually completely different or overlapping - remodularized views of the university software are needed on the demand of adding other features by either differently applying the coloring algorithm or by applying other graph algorithms, e.g., graph matching². Tab. 4.1 presents different sample representations of the university administration example as a graph.

One can easily generalize from graph algorithms to generic application logic in other domains such as e.g., price calculation logic in the domain of web-based order systems, bonus calculation and administration logic in the domain of online travel agency software, etc. Now that we have illustrated the meaning of the term generic functionality, the message we want to convey is that appropriate language technology should support a software development process in which such generic functionality as graph coloring or price calculation (a) are provided as ‘off-the-shelf’ components whose implementation is decoupled from any particular application, and (b) can be

¹A graph coloring is an assignment of colors to the vertices of a graph such that no two vertices that are connected by an edge have the same color. A minimum coloring is a coloring with a minimum number of colors.

²A matching is a subset $M \subseteq E$ of the graph’s edges such that every vertex is connected to at most one edge from M . A maximum matching is matching with a maximum number of edges in M .

Graph	Vertex	Edge (v1,v2)
Course Collision	Courses	Teacher teaches both v1 and v2 or both v1 and v2 required by same student year
Student Contacts	Students	v1 and v2 visit a common course
Student knows Teacher	Students, Teachers	Student v1 visits a course by teacher v2
Teacher uses Room	Teachers, Rooms	Course with Teacher v1 and assigned room v2

Table 4.1: Possible Mappings from University Example to Graph

integrated a-posteriori into a multitude of existing software.

The requirement for independent implementation of generic functionality calls for appropriate module constructs for doing so. The requirement for a-posteriori integration implies that we need a remodularization of the existing software, however, without physically changing it. A physical change is not only undesirable but also frequently impossible. There is in fact no single physical change of the modular structure that would satisfy the needs of all generic functionality to be integrated, since they have in general quite different views of what the modular structure should be.

In the absence of appropriate language technology, the implementation of different graph algorithms will be scattered around several classes, often duplicated, rendering the resulting software a nightmare to maintain and evolve. To use the terminology of aspect-oriented software development community, graph algorithm implementations would be scattered in and tangled with the modular structure of the university software. Unfortunately, as argued in [Ber90, Höl93, MBF99], current object-oriented languages are not very well equipped to cope with the subtle problems that occur when integrating independently developed components. Industrial component models such as EJB and CCM do not tackle this problem, either. With beans in the EJB model one can indeed ‘write application logic once and run it in (almost) any server platform’. However, the integration of generic independently developed application logic into existing EJB software is not supported.

The problem does not only apply to the integration of components from third party vendors but also to the integration of reusable modules in general: those that capture different separated concerns of a system in any software engineering effort. The principles of *separation of concerns* [Dij76] and its modern incarnation as *aspect-oriented programming* [KLM⁺97, TOHS99, AWB⁺93, ML98], tells us that we should try to divide

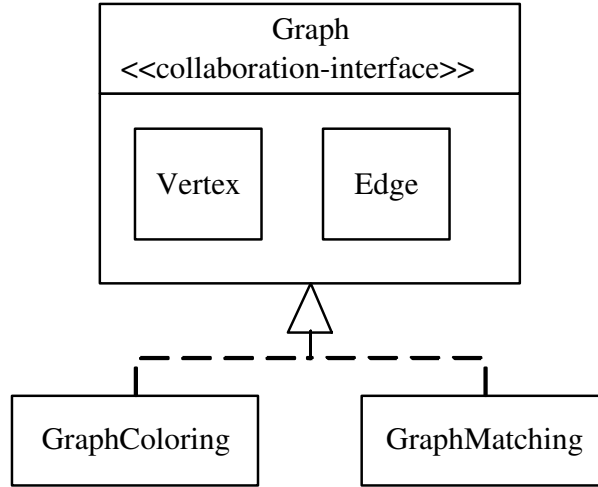


Figure 4.2: Graph collaboration

our software in smaller pieces that are as independent from each other as possible, in order to facilitate maintenance, understandability and reusability (see also Chap. 1). However, as indicated in [Kic01], the aspect-oriented programming community recognizes that still much has to be done for supporting flexible integration of crosscutting concerns.

The work presented in this chapter aims at improving the state-of-the-art technologies targeted at the problem domain outlined so far. To support independent implementation of generic functionality, we introduce *collaboration interfaces* for declaring generic component types. Collaboration interfaces differ in two ways from standard interfaces as we know them, e.g., from Java.

First, they can be nested, thereby allowing the bundling of several abstractions that together build up the concept world of a component type. For example, a component that provides algorithms on graphs articulates its world outlook - the structure and the requirements of a graph to which the algorithms could be applied - in the collaboration interface. Other components that also operate on graphs can refer to the same collaboration interface. This is schematically illustrated in Fig. 4.2. Second, in addition to expressing what a client can expect from an implementation of the interface - the *provided contract* -, collaboration interfaces also explicitly capture what interface implementations expect from potential client contexts in which they might be integrated. We say, they also make explicit the *expected contract*.

In order to support flexible a-posteriori integration of generic components into existing applications, we distinguish between *implementing* and *binding* a collaboration interface. *Implementing a collaboration interface* means

implementing its provided contract, while *binding a collaboration interface* means implementing its expected contract. Binding a collaboration interface is done with respect to a particular application into which the component gets integrated. We assume that the world of the particular application is, in general, very different from the component’s world. Therefore, we provide language means to express how the abstractions of a base application should be translated to the vocabulary of a particular collaboration interface. We use the term *on-demand remodularization* for this translation process to indicate two important characteristics of our remodularization concept. First, remodularizations in our model are virtual, meaning that the base module structure is never changed physically; a remodularization rather defines a virtual view on top of the physical structure. Second, the remodularization specified for binding a collaboration interface C is effective only on the demand of executing functionality in C . In other words, the semantics of existing programs remains unchanged as long as the remodularization is not explicitly applied.

Please note that our use of the term is not identical to its use in the context of HyperJ [OT00], where it was originally introduced. We will explain the difference later.

An important insight that drove our approach to linguistic means to express bindings is that simple mappings from component abstractions to base classes (“In the collaboration C , class X plays the role R ”) are not sufficient. The base application does not necessarily have classes that directly correspond to a role in a particular collaboration. For instance, there is no abstraction in the university software that directly corresponds to the edge abstraction in the course collision graph (Tab. 4.1). Edges are only implicitly represented as pairs of courses that need to be computed at runtime. Hence, our view that mapping abstractions from the two worlds needs full computational power, which is usually not provided by declarative mapping constructs. One of the contributions of this model is the fact that we present an approach which allows such flexible mappings.

In addition, our on-demand remodularization is object-based rather than class-based. Class-based means that a remodularization which affects a class applies to all instances of a class, whereas object-based means that the remodularization may be created for individual objects on-demand. The advantage of object-based remodularization is twofold. First, we have fine-grained control over the integration process because we can choose for each object whether it should be part of a collaboration or not. Second, the same object (or set of objects) can participate in multiple component instances. For example, a particular course instance can be a vertex in the course collision graph and simultaneously an edge in the “Teacher uses Room” graph (see Tab. 4.1). It can even be a vertex in another course collision graph which is independent from the first one. Hence, object-based remodularization enables us to create multiple independent remodularizations of the

same objects. This is indicated in Fig. 4.3, which outlines the general architecture for using our proposal, by two different remodularizations of the same structure. This was an important requirement on the integration of independent components (recall the different views of the university example in Tab. 4.1).

An important feature of the model is the loose coupling of the implementation and binding modules which allows to reuse them independently. The implementation of a component type relies on the declarations in the required contract in order to remain oblivious of the potential contexts of use. This renders a component implementation independent of specific applications. By having the expected contract be an integral part of its type, any component implementation carries around a port that makes it pluggable into unknown worlds. Any binding of the collaboration interface can serve as a plug. On the other side, a binding module can also rely on the declarations in the provided contract, remaining oblivious of any potential implementation of the component type being bound. However, by carrying around the provided contract of their type, they can easily be plugged with arbitrary implementations of that type.

To our best knowledge, this approach is the first one that decouples the component implementations and remodularizations as indicated in Fig. 4.3 and thus allows us to combine arbitrary implementations with arbitrary bindings of a collaboration interface. It is only after the composition with a binding module that an implementation becomes operative. The gain is that one can write code that is polymorphic with respect to either a component's implementations, or bindings, or both of them, depending on whether the code is written to a certain binding type, a certain implementation type, or to the component type, respectively. In addition, due to an appropriate generalization of common OO concepts (such as types, subtype polymorphism and late binding) from the level of individual classes to the level of sets of collaborating classes, reuse is very naturally supported in both dimensions, component implementation and remodularization.

The remainder of this chapter is organized as follows. Sec. 4.2 presents shortcomings of current language technology with respect to supporting integration of generic components. Sec. 4.3 introduces our notions of collaboration interfaces and on-demand remodularization, Sec. 4.4 elaborates on the dimensions of reusability in this approach. Sec. 4.6 discusses related work, and Sec. 4.7 summarizes the chapter.

4.2 Problem Statement

In this section, we set up the stage for the rest of the chapter. The goal is to identify shortcomings of current object-oriented language technology with respect to supporting the development of generic components designed

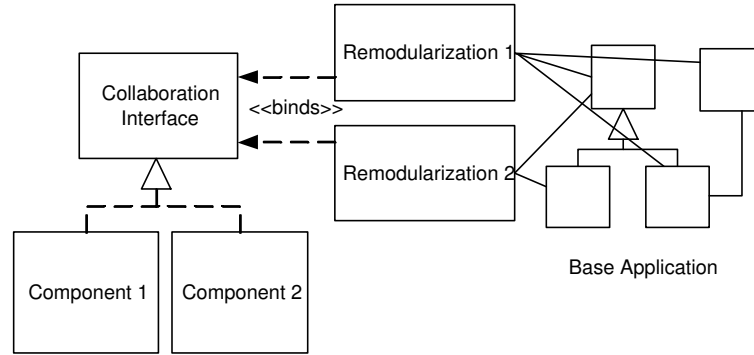


Figure 4.3: General architecture for collaboration interfaces and on-demand remodularization

for late integration into various contexts of use. The following sections will present our proposal for coping with these shortcomings. The target of our criticism is the common concept of interfaces as we know it, e.g., from Java. We argue that they lack two important features:

- **Appropriate support for declaring component types as a set of mutually recursive types.** Defining generic components involves in general several related abstractions. We claim that current technology falls short in providing appropriate means to express the different abstractions and their respective features and requirements that are involved in a particular collaboration.
- **Support for bidirectional communication:** Interfaces provide clients with a contract as what to expect from a server object that implements the interface. We say, they express the *provided* contract. In order to define generic components which are decoupled from their potential contexts of use, expressing expectations that a server might have on potential contexts of use is as important. We use the term *expected* contract to denote these expectations. What is needed is support for a loose coupling of client and server, that is (a) decoupling them to facilitate reuse, while (b) enabling them to tightly communicate with each other as part of a whole.

To illustrate these shortcomings, let us have a critical look at a simple example. Fig. 4.4 shows a simplified version of the `TreeModel` interface in Swing³, Java’s GUI framework [JFC]. This interface provides a generic description of the data model for a `JTree`, or other GUI tree controls. For

³Swing separates our interface into two interfaces, `TreeModel` and `TreeCellRenderer`. However, this is irrelevant for the reasoning in this model.

```
interface TreeModel {
    Object getRoot();
    Object[] getChildren(Object node);
    String getStringValue(Object node, boolean selected,
        boolean expanded, boolean leaf, int row, boolean focus);
}

interface TreeGUIControl {
    display();
}

class SimpleTreeDisplay implements TreeGUIControl {
    TreeModel tm;
    display() {
        Object root = tm.getRoot();
        ... tm.getChildren(root) ...
        ...
        // prepare parameters for getStringValue
        ... tm.getStringValue(...);
        ...
    }
}
```

Figure 4.4: Simplified version of the Java Swing `TreeModel` interface

illustration purposes, Fig. 4.4 also presents a pseudo interface for tree GUI controls in `TreeGUIControl`, as well as a pseudo implementation of this interface in `SimpleTreeDisplay` (the latter roughly corresponds to `JTree`).

In our terminology the code in Fig. 4.4 defines a generic component for displaying arbitrary data structures that can be viewed as trees in a GUI. When this component is used in a particular context, e.g., for object structures that represent arithmetic expressions, it provides to this context the `display` functionality. In turn, it expects from the context a concrete implementation of `getChildren` and `getStringValue`. These operations can only be implemented specifically for a concrete data type to be presented as a tree. That is, `TreeGUIControl` corresponds roughly to what we called the provided contract in the type of our component, while `TreeModel` corresponds roughly to what we called the expected contract. The class `SimpleTreeDisplay` represents a sample implementation of the provided contract.

The design in Fig. 4.4 does actually a good job in decoupling these two contracts. Different implementation of GUI controls can be written to the `TreeModel` interface and can therefore be reused with a variety of concrete

```
class Expression {  
    Expression[] subExpressions;  
    String description() { ... }  
    Expression[] getSubExpressions() { ... }  
}  
class Plus extends Expression { ... }
```

Figure 4.5: Expression Trees

implementations of it, i.e., with a variety of data structures. The other way around, any data structure to be displayed is decoupled from a specific tree GUI control (e.g., `JTree`), such that the data structure can be displayed with different GUI tree controls.

So, what is wrong with the approach to specifying generic components exemplified by the design in Fig. 4.4? The first bad smell is the frequent occurrence of the type `Object`. We know that a tree abstraction is defined in terms of smaller tree node abstractions. However, this collaboration of the tree and tree node abstractions is not made explicit in the interface. Since the interface does not state anything about the definition of tree nodes, it has to use the type `Object` for nodes.

The disadvantages of using the most general type, `Object`, are twofold. First, it is conceptually questionable. If every abstraction that is involved in the component definition is only known as `Object`, no messages, beside those defined in `Object`, can be directly called on those abstractions. Instead, a respective top-level interface method has to be defined, whose first parameter is the receiver in question. For example, the methods `getChildren` and `getStringValue` conceptually belong to the interface of a tree node, rather than of a tree. Since the tree definition above does not include the declaration of a tree node, they are defined as top-level methods of the tree abstraction whose first argument is `Object node`.

Second, we lose type safety. Let us have a look at Fig. 4.5 and Fig. 4.6. Fig. 4.5 shows a simple base application for expressions, and Fig. 4.6 demonstrates how the expression classes can be adapted ('remodularized') to fit in the conceptual world of a `TreeModel`. In our terminology, `ExpressionDisplay` in Fig. 4.6 represents an implementation of the *expected* contract. Since we use `Object` all the time, we cannot rely on the type checker to prove our code statically safe because type-casts are ubiquitous.

The question naturally raises here: Why didn't the Swing designers define an explicit interface for tree nodes as in Fig. 4.7 from the very beginning? Well, there are good reasons for this. With the explicit type `NodeTree` it becomes more difficult to decouple the two contracts, i.e., the data structures to be displayed from the display algorithm. The idea is that the wrapper classes around e.g., `Expression` would look like in Fig. 4.8. The problem

```
class ExpressionDisplay implements TreeModel {
    ExpressionDisplay(Expression r) { root = r; }
    Expression root;
    Object getRoot() { return root; }
    Object[] getChildren(Object node) {
        return ((Expression) node).getSubExpressions();
    }
    String getStringValue(Object node, boolean selected,
        boolean expanded, boolean leaf, int row, boolean focus){
        String s = ((Expression) node).description();
        if (focus) s = "<"+s+">";
        return s;
    }
}
```

Figure 4.6: Using `TreeModel` to display expressions

```
interface TreeDisplay {
    TreeNode getRoot();
}
interface TreeNode {
    TreeNode[] getChildren();
    String getStringValue(boolean selected,
        boolean expanded, boolean leaf, int row, boolean focus);
}
```

Figure 4.7: `TreeDisplay` interface with explicitly reified `TreeNode` interface

with such kind of wrappers, as also indicated by Hölzle [Höl93], is that we create new wrappers every time we need a wrapper for an expression. This leads to the *identity hell*: we loose the state and identity of previously created wrappers for the same node. The questionable alternative would be to use hash tables which is not only laborious but does also involve the definition and use of additional classes for maintaining these hashtables, thereby rendering the code more complex and less readable⁴.

So far, we discussed problems resulting from the lack of appropriate support for defining multiple related abstractions in one module. Let us now illustrate the problems resulting from the second shortcoming of standard interfaces: missing support for bidirectional communication. Consider for this purpose the `getStringValue()` method in Fig. 4.4 and Fig. 4.6. This method has noticeable many parameters that might be of interest when

⁴In fact, Swing offers a `TreeNode` interface similar to the one in Fig. 4.7. However, classes that define data structures to be displayed as tree nodes should anticipate this and explicitly implement the interface.

```

class ExprAsTreeNode
implements TreeNode {
    Expression expr;
    void getStringValue(...) {
        // as before
    }
    TreeNode[] getChildren() {
        Expressions[] subExpr = expr.getSubExpressions();
        TreeNode[] children =
            new TreeNode[subExpr.length];
        for (i = 0; i<subExpr.length; i++) {
            children[i] = new ExprAsTreeNode(subExpr[i]);
        }
        return children;
    }
}

```

Figure 4.8: Mapping `TreeNode` to `Expression`

computing a string representation of the node. *Might be.* The sample implementation in Fig. 4.6 uses only the `selected` parameter and ignores the others. That means, the tree GUI control, which calls this method on the `TreeModel` interface, has to perform expensive computations to obtain the parameter values for this method (see implementation of `SimpleTreeDisplay::display()` in Fig. 4.4), although they might be rarely all used.

This is a typical case where we would like to establish a bidirectional communication between the two contracts of the tree displaying component. Here we would like `ExpressionDisplay.getStringValue` to explicitly ask the tree GUI control to compute only relevant values for it, like `selected` or `hasFocus`, implying the GUI control interface provides respective operations. Recall that the GUI control interface corresponds to the provided interface of our generic component for displaying arbitrary data structures that can be viewed as trees in a GUI. As for now, the interfaces are completely separated (into `TreeModel` and `TreeGUIControl`), and there is nothing in the design that would suggest their tight relation as two faces of the same abstraction. As such, there is no build-in support for bidirectional communication between their respective implementations. Build-in means by the virtue of implementing two faces of the same abstraction, which serves as the implicit communication channel.

One can certainly achieve the desired communication by additional infrastructure (e.g., via cross-references) which has to be communicated to the respective programmers. However, we think that bidirectional communication is such a natural and frequent concept that the overhead that is necessary to enable bidirectional communication with conventional interfaces is too high. Please note that the additional `TreeNode` interface would also be

of no help concerning the bidirectional communication problem exemplified by the `getStringValue()` method.

The third point is the fact that it is difficult and awkward to associate state with abstractions like our tree nodes. We might want to associate state with tree nodes in both the `ExpressionDisplay` class in Fig. 4.6 and also inside the tree GUI control. For example, we might want to cache the computed string value or children in Fig. 4.6, because the re-computation might be expensive. In the GUI control itself, we might want to associate state like whether a tree node is selected or not or its position on the screen with the respective tree node. The only means to associate state with tree nodes is to make extensive use of hash tables, which is laborious and awkward.

4.3 Core Concepts

In this section, we will give an overview of the concepts that comprise our model by means of the `TreeDisplay` example from the previous section.

4.3.1 Collaboration Interfaces, their Implementations and Bindings

In order to cope with the problems discussed in Sec. 4.2 we propose the notion of *collaboration interfaces* (*CI* for short), which differ from standard interfaces in two ways. First, CIs introduce the **provided** and **required** modifiers to annotate operations belonging to the provided and the expected contracts, respectively, hence supporting bidirectional interaction between clients and servers. Second, CIs exploit interface nesting in order to express the interplay between multiple abstractions participating in the definition of a generic component.

For illustration, the CI `TreeDisplay` that bundles the definition of the generic tree displaying functionality from Sec. 4.2 is shown in Fig. 4.9. As an example for the provided and expected contract, consider the methods `TreeDisplay.display()` and `TreeDisplay.getRoot()` in Fig. 4.9. Any tree display object is able to display itself on the request of a client - hence the **provided** modifier for `TreeDisplay.display`. However, in order to do so, it expects a client specific way of how to access the root tree node. What the root of a displayable tree will be depends on (a) which modules in a concrete deployment context of `TreeDisplay` will be seen as tree nodes and, (b) which one of them will play the role of the root node. Hence, the declaration of `getRoot` with the expected modifier. `TreeDisplay` comes with its own definition of a tree node: The CI `TreeNode` is nested into the declaration of `TreeDisplay`. Please note that nesting of bidirectional interfaces in our approach has a much deeper semantics than usual nested classes and interfaces in Java: the nested interfaces are namely *virtual types* as in [Ern01]. We will elaborate on that in Sec. 4.3.4.

```

interface TreeDisplay {
    provided void display();
    expected TreeNode getRoot();

    interface TreeNode {
        expected TreeNode[] getChildren();
        expected String getStringValue();
        provided display();
        provided boolean isSelected();
        provided boolean isExpanded();
        provided boolean isLeaf();
        provided int row();
        provided boolean hasFocus();
    }
}

```

Figure 4.9: Collaboration interface for TreeDisplay

The categorisation of the operations into expected and provided comes with a new model of what it means to implement an interface. We explicitly distinguish between *implementing* an interface’s provided contract and *binding* the same interface’s expected contract. Two different keywords are used for this purpose: **implements**, respectively **binds**. In the following, we refer to classes that are declared with the **implements** keyword as *implementation classes*. Similarly, we refer to classes that are declared with the **binds** keyword as *binding classes*.

An implementation class of a CI must (a) implement all **provided** methods of the CI and (b) provide an implementation class for each of the CI’s nested interfaces. In doing so, it is free to use respective **expected** methods. In addition, an implementation class may or may not add additional methods and state to the CI’s abstractions it implements. Fig. 4.10 shows a sample tree GUI control that implements **TreeDisplay**. The class **SimpleTreeDisplay** implements the only provided operation of **TreeDisplay**, **display()**, by forwarding to the result of calling the expected operation **getRoot()**. In addition to implementing **display()**, **SimpleTreeDisplay** must also provide a nested class that implements **TreeNode** - the only nested interface of **TreeDisplay**. The correspondence between a nested implementation class and its corresponding nested interface is based on name identity – **SimpleTreeDisplay** e.g., defines a class named **TreeNode** which is the implementation of the nested interface with the same name in **TreeDisplay**. This nested class has to implement all **provided** methods of the **TreeNode** interface, e.g., **display()**. The declaration of the instance variable **boolean selected** and the corresponding query operation **isSelected** in **SimpleDisplay.TreeNode** are examples of new declarations added by an


```
class SimpleTreeDisplay implements TreeDisplay {
    void onSelectionChange(TreeNode n, boolean selected) {
        n.setSelected(true);
    }
    void display() {
        getRoot().display();
    }

    class TreeNode {
        boolean selected;
        ...
        boolean isSelected() { return selected; }
        // other provided methods similar to selected
        void setSelected(boolean s) { selected =s;}
        void display() {
            ... TreeNode c = getChildren()[i];
            ... paint(position, c.getStringValue());
            ...
        }
    }
}
```

Figure 4.10: A sample implementation of `TreeDisplay`

implementation class. Please note that just as nested interfaces, all nested implementation classes are virtual types (see Sec. 4.3.4).

A binding class of a CI must (a) implement all **expected** methods of the CI, and (b) provide zero or more binding classes for each of the CI's nested interfaces (we may have multiple bindings of the same interface, see subsequent discussion). Just as implementation classes can use their respective expected facets, the implementation of the expected methods of a CI and its nested interfaces can also call methods declared in the respective provided facets. The process of binding a CI instantiates its nested types for a concrete usage scenario of the generic functionality defined by the CI. Hence, it is natural that in addition to their provided facets, binding classes also use the interface of abstractions from that concrete usage scenario. We say that bindings wrap abstractions from the world of the concrete usage scenario and map them to abstractions from the generic component world.

For illustration, the class `ExpressionDisplay` in Fig. 4.11 shows an example of binding the generic `TreeDisplay` CI from Fig. 4.9 for the concrete usage scenario, in which `Expression` structures are to be viewed as the trees to display. First, `ExpressionDisplay` binds the nested type `TreeNode` as shown in the nested class `ExprTreeNode`. The latter implements all expected methods of `TreeNode` by using (a) the provided facet of `TreeNode`, and (b) the interface of the class `Expression` (via the instance

```

class ExpressionDisplay binds TreeDisplay {
    Expression root;

    public ExpressionDisplay(Expression rootExpr) {
        root = rootExpr;
    }

    TreeNode getRoot() {
        return ExprTreeNode(root);
    }

    class ExprTreeNode binds TreeNode {
        Expression e;
        ExprTreeNode(Expression e) { this.e=e;}
        TreeNode[] getChildren() {
            return ExprTreeNode[](e.getSubExpressions());
        }
        String getStringValue() {
            String s = e.description();
            if (hasFocus()) s = "<"+s+">";
            return s;
        }
    }
}

```

Figure 4.11: Binding of `TreeDisplay` for expressions

variable `e`). Consider e.g., the implementation of the method `ExprTreeNode.getStringValue()`, which calls both `TreeNode.hasFocus()` as well as `Expression.getDescription()`.

In addition to binding `TreeNode`, `ExpressionDisplay` also implements the method `getRoot()` - the only method declared in the *expected* facet of `TreeDisplay`. Here is where the reference `root` to the `Expression` object to be seen as the root of the expression structure to display is transformed into a `TreeNode` by being wrapped into an `ExprTreeNode` object. Please note that this wrapping does not happen via an ordinary constructor call - `new ExprTreeNode(root)` in this case -, but rather by means of the *wrapper recycling* call `ExprTreeNode(root)`. We will elaborate on the concept of *wrapper recycling* in a moment.

Except for binding the interface `TreeNode`, `ExprTreeNode` is basically a usual class that, in this case, wraps an instance of `Expression`. Since wrapping of objects in these classes is a very common task, we add some syntactic sugar for the most common case, namely by a `wraps` clause.

The semantics of `wraps` is that

```
class ExprTreeNode binds TreeNode wraps Expression {...}
```

is equivalent to

```
class ExpressionDisplay binds TreeDisplay {  
  ...  
  class ExprTreeNode binds TreeNode wraps Expression{  
    TreeNode[] getChildren() {  
      return ExprTreeNode[] (wrappee.getSubExpressions());  
    }  
    String getStringValue() {  
      String s = wrappee.description();  
      if (hasFocus()) s = "<"+s+">";  
      return s;  
    }  
  }  
}
```

Figure 4.12: Alternative encoding of ExprTreeNode using the wraps clause

```
class ExprTreeNode binds TreeNode {  
  Expression wrappee;  
  ExprTreeNode(Expression e) { wrappee = e;}  
  ... }
```

Using `wraps`, the code in Fig. 4.11 can be rewritten as in Fig. 4.12. In the following code we will make frequent use of `wraps` but it is important to understand that it is just syntactic sugar and does not prevent us to create arbitrarily complex initialization procedures by using ordinary constructors.

The careful reader should have noticed that we do not use identical names for establishing the correspondence between a binding class and its corresponding nested interface, as we did with implementing classes (`ExprTreeNode` in Fig. 4.11 binds `TreeNode`, but is itself not called `TreeNode`). As indicated in Sec. 4.1, different bindings of the same interface might be needed, if we have different abstractions in the concrete usage scenario that have to be mapped to the same component abstraction. This was illustrated by the “Student knows Teacher” graph from Tab. 4.1: Both, students and teachers, play the role of vertices in this graph. To cope with such multiple bindings of the same interface, the identification by name that we had with implementation classes is not carried over to binding classes. This enables multiple different bindings of the same component type to different base types without running into conflicts, since the bindings can still be discriminated by their names.

The careful reader might notice that expected methods in a collaboration interface are similar to the abstract primitive operations in the template method pattern [GHJV95]. There are two significant differences, though. First, CIs describe the interplay between multiple abstractions whereas the template method pattern deals only with a single abstraction. Second, while the implementation of the template methods can be reused with different bindings of the primitive operations, the other way around is not possible.

Each binding of the primitive operations is bound to a single implementation of the template methods. On the contrary, CI binding and implementations are separated in independent modules and related to each other by being pieces of the overall implementation of the same common CI. As such, they can be reused independently.

In Sec. 4.1 we gave an example that illustrated why declarative mapping constructs as in [TOHS99, ML98, MSL01] are not sufficient to express arbitrary on-demand remodularizations. In general, the full computational power of an object-oriented language is needed for this purpose. For this reason, our approach to specifying remodularizations is rather manual. In fact, binding classes and their nested classes are almost standard classes. *Almost* stands for two differences. First, nested binding classes are also virtual types (see Sec. 4.3.4). Second, they make use of the notion of *wrapper recycling*, which we discuss next.

4.3.2 Wrapper Recycling

Wrapper recycling is our mechanism to escape the wrapper identity hell mentioned in Sec. 4.2. It is a concept of how to create and maintain wrapper instances, and a way to navigate between abstractions of the component world and abstractions of the base world - the concrete usage scenario world -, ensuring that the same (identical) wrapper instance will always be retrieved for a set of constructor arguments. This way the state and the identity of the wrappers is preserved.

Syntactically, wrapper recycling refers to the fact that, instead of creating an instance of a wrapper `W` with a standard `new W(constructorargs)` constructor call, a wrapper is retrieved with the construct `outerClassInstance.W(constructorargs)`. For illustration consider once again the expression `return ExprTreeNode(root)` in the method `ExpressionDisplay.getRoot()` in Fig. 4.11. We already mentioned in the previous section that the expression in the return statement is not a standard constructor call, but rather a wrapper recycling operator. We use the usual Java scoping rules, i.e., `return ExprTreeNode(root)` is just an abbreviation for `return this.ExprTreeNode(root)`.

The idea is that we want to avoid creating a new `ExprTreeNode` wrapper each time the method `getRoot()` is called on an `ExpressionDisplay`. The call to the wrapper recycling operation `ExprTreeNode(root)` is equivalent to the corresponding constructor call, only if a wrapper for `root` does not already exist, ensuring that there is a unique `ExprTreeNode` wrapper for each expression within the context of the enclosing `ExpressionDisplay` instance. That is, two subsequent wrapper retrievals for an expression `e` yield the same wrapper instance - the identity and state of the wrapper are preserved.

This is due to the semantics of a wrapper recycling call, which is as follows: The outer class instance maintains a map `mapW` for each nested wrapper

class `W`. An expression `outerClassInstance.W(wrapperargs)` corresponds to the following sequence of actions:

1. Create a compound key for the constructor arguments, lookup this key in `mapW`.
2. If the lookup for the key fails, create an instance of `outerClassInstance.W` with the annotated constructor arguments, store it in the hash table `mapW`, and return the new instance. Otherwise return the object already stored in `mapW` for the key.

The wrapper recycling call `ExprTreeNode[](...)` in the method `ExprTreeNode.getChildren` in Fig. 4.11 is an example for the syntactic sugar we use to express wrapper recycling of arrays, namely an automatic retrieval of an array of wrappers for an array of base objects.

A naive implementation of wrapper recycling in a language with garbage collection would imply a memory hole because wrapped objects would never be collected by the garbage collector. However, this can easily be reconciled by more advanced memory management techniques such as weak references and reference queues. Java, for example, has a standard API class `WeakHashMap` that could be used instead of a usual map.

4.3.3 Composing Bindings and Implementations

Both classes defined in Fig. 4.10 and 4.11 are not operational, i.e., cannot be instantiated, even if they are not annotated as abstract. These classes are indeed not abstract, since they are complete implementations of their respective contracts. The point is that the respective contracts are parts of a whole and make sense only within a whole. Operational classes that completely implement an interface are created by composing an implementation and a binding class, syntactically denoted as *aCollabIfc<aBinding, anImpl>*. This is illustrated by the class `SimpleExpressionDisplay` in Fig. 4.13, which declares itself as an extension of the composed class `TreeDisplay<SimpleTreeDisplay, ExpressionDisplay>`. Only such compound classes are allowed to be instantiated by the compiler. For instance, Fig. 4.13 also shows sample code that instantiates and uses the compound class `SimpleExpressionDisplay`.

Combining two classes as in Fig. 4.13 means that we create a new compound class within which the respective implementations of the **expected** and **provided** methods are combined. The same combination also takes place recursively for the nested classes: All nested classes with a **binds** declaration are combined with the corresponding implementation from the component class. The separation of the two contracts, their independent implementation, and the dedicated late composition, allows us to freely reuse implementations of the two contracts in arbitrary compositions. We could

```

class SimpleExpressionDisplay extends
    TreeDisplay<SimpleTreeDisplay,ExpressionDisplay> {}
...

Expression test = new Plus(new Times(5, 3), 9);
TreeDisplay t = new SimpleExpressionDisplay(test);
t.display();

```

Figure 4.13: Creating and using compound classes

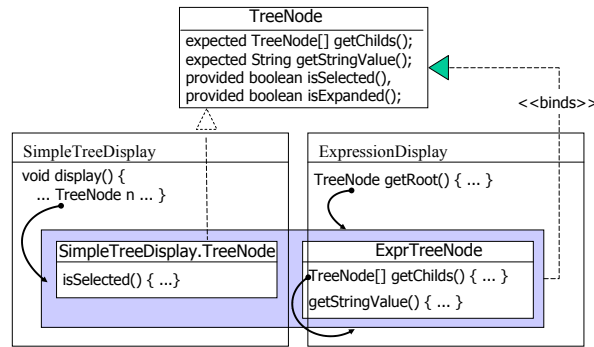


Figure 4.14: Type rebinding in compound classes

combine `SimpleTreeDisplay` with any other binding of `TreeDisplay`. Similarly, `ExpressionDisplay` could be combined with any other implementation of `TreeDisplay`. Sec. 4.4 will have more to say about the reuse dimensions and the flexibility supported by our approach.

Note that the overall definition of the nested type, e.g., `TreeNode`, depends on the concrete composition of implementation and binding types within which the type is used. This does not only affect the external clients, but also the internal references. For instance, references to `TreeNode` within `ExpressionDisplay` and `SimpleDisplay` are rebound to the composed definition of `TreeNode` in `SimpleExpressionDisplay`, as illustrated in Fig. 4.14. Their meaning would be different in another compound class, e.g., resulting from composing `SimpleDisplay` with another binding class, or `ExpressionDisplay` with another implementation class. This is a natural consequence of the fact that nested types introduced by the collaboration interfaces are virtual types, on which we will elaborate in the following.

4.3.4 Virtual Types

In CAESAR approach, all types that are declared as nested interfaces of a CI and all classes that implement or bind such interfaces (including classes

that extend the latter) are *virtual types* and *virtual classes*, respectively [MMP89]. In the context of this thesis, we use the notion of virtual types of the family polymorphism approach [Ern01]. This means: (a) similar to fields and methods, types also become properties of objects of the class in which they are defined, and consequently (b) their denotation can only be determined in the context of an instance of the enclosing class. Hence, the meaning of a virtual type is late bound depending on the receiver object that executes when the virtual type at hand is referenced.

Consequently, all type declarations, constructor calls, and wrapper recycling calls for virtual types/classes within a CI are actually always annotated with an instance of the enclosing class. That is, type declarations and constructor invocations are always of the form `enclInst.MyVirtual x`, respectively `enclInst.MyConstructor()`. Similarly, wrapper recycling calls are also always of the form `outerClassInstance.W(args)` and not simply `W(args)`. For the sake of simplification, we apply the scoping rules common for Java nested classes also to type declarations and constructor or wrapper recycling calls: A call `OuterClass.this.W(args)` can be shortened to `W(args)`, and the type declaration `OuterClass.this.W` can be shorted to `W` as long as there are no ambiguities. This scoping rule applies to all type declarations and wrapper recycling calls that have appeared so far in this chapter.

For instance, all references to `ExprTreeNode` in Fig. 4.11 should be read as `ExpressionDisplay.this.ExprTreeNode`. The implication is that the meaning of any reference to the type name `ExprTreeNode` within the code of `ExpressionDisplay` will be bound to the compound class that combines `ExpressionDisplay.ExprTreeNode` with the implementation class of `TreeNode` that is appropriate in the respective execution context. For example, in the context of a `SimpleExpressionDisplay` as in Fig. 4.13, `ExprTreeNode` will be bound to the respective definition in the compound class `TreeDisplay<ExpressionDisplay,TreeNode>`. The same references will be bound differently if they occur in the execution context of an object of some subclass of `ExpressionDisplay` or in the context of a different implementation class. The same also applies to nested implementation and compound classes.

The rationale behind using virtual types lies in their power with respect to supporting reuse and polymorphism, as argued in [Ern01]. The advantages with respect to the degree of reuse we gain in our context will be discussed in Sec. 4.4. At this point, we will rather shortly discuss how our specific use of virtual types (borrowed from [Ern01]) does *not* suffer from covariance problems usually associated with virtual types, as for example the virtual type proposal in [Tho97], which requires runtime type checks. If we have a virtual type in a contravariant position, as for example the argument type of `setRoot` in Fig. 4.15, type safety is still preserved, because subsumption is disallowed if the enclosing instances are not identical. In or-

```

Expression e = ...;
final ExpressionDisplay ed =
    new SimpleExpressionDisplay(e);
...
// let FileSystemDisplay be a binding of
// TreeDisplay to the file system structure
class SimpleFileSystemDisplay extends
    TreeDisplay<SimpleTreeDisplay,FileSystemDisplay> {};

FileSystem fs = ... ;
final FileSystemDisplay fsd =
    new SimpleFileSystemDisplay(fs);
...
ed.TreeNode t = ed.getRoot();
fsd.setRoot(t); // Type error detected by typechecker!
                // sd.TreeNode is not subtype of ed.TreeNode

```

Figure 4.15: Type safety due to family polymorphism

der to make the approach sound, all variables that are used as part of type declarations have to be declared as **final** because otherwise the meaning of a type declaration might change due to a field update. For illustration consider the declaration of the variable **ed** in the sample code in Fig. 4.15. It is used as part of a type declaration for the variable **t** and is therefore declared as **final**. For more details on typing issues we refer to [Ern01].

4.3.5 Object Constructors

Having classes with splitted code, as in our division into a binding part and an implementation part, the question of object construction and constructor calls arises. Which site (binding or implementation) should be able to implement, respectively call, constructors? An important prerequisite in the following discussion is that we assume that – in general – constructors have arguments.

Allowing both sites to implement and call constructors would be unsound because every site would only call its own constructors (it does not know about the existence of the other-side constructors), and therefore invariants that are established in the constructors of the other site will not hold since the constructor of the other site will never be called - implicit constructor invocation is not possible if the constructor requires arguments.

Our point of view is that only the binding site should implement constructors because the binding site needs to establish links to base objects (which are adapted to the role they play in the particular collaboration of a generic component) in order to fulfill its purpose. If the implementation site needs to create objects, this can easily be done by specifying correspond-

ing **expected** factory methods in the collaboration interface which can be called from the implementation site and are implemented at the binding site. Therefore, only the binding classes can implement constructors, and these constructors are also the constructors that are available in the compound classes that combine an implementation class with a binding class.

Although this point (object creation) seems to be only marginal, it has an important conceptual implication that is related to *symmetry*. At first, collaboration interface implementation and binding seem to be rather symmetric, but object creation creates an important asymmetry. A consequence of this asymmetry is that we can have only one implementation of a bidirectional interface, but we may have multiple different bindings of a bidirectional interface; we can select among these bindings by means of different constructors.

4.3.6 Most Specific Wrappers

Another interesting feature of CAESAR is its notion of *most specific wrappers*, that is, a mechanism that determines the most specific wrapper for an object based on the object's runtime type. The basic idea is that we allow multiple nested classes with the same name that can be differentiated by their constructors. Consider, e.g., the code in Fig. 4.16. The class `ExpressionDisplay` contains three different classes with the name `ExprTreeNode` but each of these classes has a different constructor (recall that the `wraps` clause is just syntactic sugar for a corresponding constructor). If we now make a constructor- or wrapper recycling call, the runtime type of the constructor argument determines the actual implementation which is instantiated/recycled.

For example, in class `Test` in Fig. 4.16, the method `printStrinvValue()` has a parameter of type `Expression`. If this method would be called with an instance of `Plus`, the wrapper recycling call `ed.ExprTreeNode(e)` would yield an instance of the `ExprTreeNode` implementation that wraps `Plus`, hence “+” would be printed.

This mechanism is very similar to method dispatch in multiple dispatch languages such as CLOS, Cecil [Cha92], or MultiJava [CLCM00]. More precisely, if one thinks of the constructors of nested classes as factory methods of the enclosing instance, then our mechanism for most specific wrappers is an application of multiple dispatch at these factory methods.

4.3.7 Interim Evaluation of the Model

As an interim result, let us compare the way the generic tree display functionality and its instantiation for expressions was modeled with our model to the conventional solution discussed in Sec. 4.2.

```
class ExpressionDisplay binds TreeDisplay {
  ...
  class ExprTreeNode binds TreeNode wraps Expression{
    String getStringValue() { return null; }

    class ExprTreeNode binds TreeNode wraps Plus {
      String getStringValue() { return "+"; }
    }

    class ExprTreeNode binds TreeNode wraps Num {
      String getStringValue() {
        return Integer.toString(wrappee.getValue());
      }
    }
    ...
  }
}

class Test {
  ExpressionDisplay ed = ...;
  void printStringValue(Expression e) {
    System.out.println(
      ed.ExprTreeNode(e).getStringValue());
  }
}
```

Figure 4.16: Using most specific wrappers

- Other than the Swing interface in Fig. 4.4, we do not need to use `Object`; every item is well-typed and we do not need type casts. The methods that are conceptually part of the interface of tree nodes, are expressed as methods of a dedicated nested interface.
- Due to bidirectional interfaces, we do not have the problem related to the `getStringValue()` parameters: The implementation of this method, as in Fig. 4.11, causes the computation of only those values about the state of displaying that are really needed by means of calling appropriate methods in the `provided` interface.
- It is easy to associate additional state with tree nodes. For example, the `TreeNode` implementation in Fig. 4.10 adds a `selected` field, and the `TreeNode` binding in Fig. 4.11 could as well have added extra state to `ExprTreeNode`.

4.4 Dimensions of Reuse

In this section we want to elaborate on the degrees of reuse and polymorphism supported by our proposal. For this purpose, we will use the graph

example from Sec. 4.1, since it is better suited to demonstrate the advantages of our approach.

4.4.1 Component Type Hierarchies

The first kind of reuse supported by our model is along the dimension of component types. The key object-oriented notion of subtyping between individual interfaces extends very naturally to our nested collaboration interfaces. New CIs can be defined as extensions of already existing CIs via the **extends** clause. The new CI inherits all nested type definitions and provided/expected methods of its parent CI. The inheriting CI can then add new nested type definitions and expected/provided method declarations. In addition, the inherited nested types can be refined by defining interfaces with the same name annotated by the modifier **override**. An “overriding” nested type inherits all declarations of the nested type being overridden and can add new declarations.

For illustration, Fig. 4.17 shows three sample collaboration interfaces for graphs. The top interface **Graph** defines the general graph abstractions and properties of these abstractions. The other two interfaces, **ColoredGraph** and **MatchedGraph**, refine **Graph** by adding methods or refining inherited nested interfaces of **Graph**⁵.

The refinement of nested types has pretty much the semantics of standard inheritance on types. So, why the new syntax - the keyword **override** rather than the familiar **extends**? The reason becomes clear once you recall that our nested types are virtual types, rather than standard types as e.g., Java interfaces. Other than an ordinary **extends** declaration, an **override** declaration does not create a *new* type with a *new* name but overrides the definition of the inherited type. The typing implications of these virtual types were explained in the previous section.

4.4.2 Implementation Hierarchies

Fig. 4.18 shows two implementation classes for the **ColoredGraph** interface from Fig. 4.17, **SuccessiveAugmentationColoring** and **SimulatedAnnealingColoring**, each employing a different algorithm for graph coloring. By being implementation classes, each of them provides implementations for the **provided** methods of **ColoredGraph**, using the declared **expected** methods. In addition, they may add new declarations. For example, **SuccessiveAugmentationColoring** adds a field **temp_color** to **Vertex**. The association of

⁵The purpose of the **ColoredGraph.Edge.getBadness()** method is to have a measure of how troublesome a particular edge is with respect to minimum coloring, meaning that if an edge with a high badness would be removed, it is likely that we can color the graph with less colors.

```
interface Graph {
    interface Vertex {
        expected Edge[] getEdges();
    }
    interface Edge {
        expected Vertex getV1();
        expected Vertex getV2();
    }
}

interface ColoredGraph extends Graph {
    provided computeMinimumColoring(Vertex v[]);
    override interface Vertex {
        expected void setColor(int c);
        expected int getColor();
    }
    override interface Edge {
        provided float getBadness();
    }
}

interface MatchedGraph extends Graph {
    provided computeMaximumMatching(Vertex v[]);
    override interface Edge {
        expected void setMatched(boolean b);
        expected boolean isMatched();
    }
}
```

Figure 4.17: Graph collaboration interfaces

the nested classes with the corresponding nested interface in **ColoredGraph** happens by common names, as already explained in Sec. 4.3.1

Similar to interface refinement as in Fig. 4.17, it is also possible to refine implementation classes, whereby the definitions of the nested classes can again be refined with the **override** modifier. In other words, the subclassing and subtyping relations between individual classes in standard OO are naturally carried over to the implementation classes of CIs; again with the important difference that our nested classes are virtual types as explained in the previous section. For example, the commonalities between the two different coloring algorithms could be factored out into a common superclass as in Fig. 4.19.

4.4.3 Binding Hierarchies

In the following we elaborate on some advanced issues related to CI bindings that could not be appropriately demonstrated by the simple example of

```
class SuccessiveAugmentationColoring
implements ColoredGraph {
    // successive augmentation coloring algorithm
    void computeMinimumColoring(Vertex v[]) {
        // successive augmentation coloring algorithm
        ... Edge e[] = v[i].getEdges(); ...
        ... Vertex w = e[j].getV2();
        ... if (w.isLegalColor(color)) w.temp_color = color;
        ... e[n].setBadness(badness); ...
        // commit final coloring
        for (int k=0; k<v.length;k++)
            v[k].setColor(v[k].temp_color);
    }
    class Vertex {
        int temp_color;
        boolean isLegalColor(int color) {
            Vertex neighbor[] = ...;
            for (int i=0;i<neighbor.length;i++)
                if (neighbor[i].getColor() == color) return false;
            return true;
        }
    }
    class Edge {
        float badness;
        float getBadness() { return badness; }
        void setBadness(float b) { badness = b; }
    }
}

class SimulatedAnnealingColoring
implements ColoredGraph {
    // Simulated Annealing coloring algorithm
    void computeMinimumColoring(Vertex v[]) {
        ...
    }
    ...
}
```

Figure 4.18: Different Coloring Algorithms

```
abstract class AbstractColoring
implements ColoredGraph {
    class Vertex { ... }
    class Edge { float badness; ... }
}

class SuccessiveAugmentationColoring
extends AbstractColoring {

    // Successive augmentation coloring algorithm
    void computeMinimumColoring(Vertex v[]) { ... }
    override class Vertex { ... }
}

class SimulatedAnnealingColoring
extends AbstractColoring {

    // Simulated Annealing coloring algorithm
    void computeMinimumColoring(Vertex v[]) { ... }
    override class Vertex { ... }
}
```

Figure 4.19: Factoring out the commonalities between the coloring algorithms

the previous section. Furthermore, we discuss the extent to which reuse is enabled along the dimension of binding classes.

Recall our claim that binding a CI to a concrete application implies an *on-demand remodularization* of the application for which simple declarative mappings are insufficient. With the more sophisticated graph example, we are now able to illustrate, how our proposal copes with this requirement. For this purpose, Fig. 4.20 shows a binding class that transforms the scheduling graph structure hidden within the university class structure to the class structure that is required by our graph algorithms. The nested classes `CourseCollision` and `CourseVertex` are remodularization wrappers around base objects. The class `CourseCollision` implements the expected interface of `ColoredGraph.Vertex` by wrapping an object `c` of type `Course`, while `CourseVertex` implements the expected interface of `ColoredGraph.Edge` by wrapping two objects of type `Course`, `c1` and `c2`.

Please note that the class `CourseCollision` wraps *two* courses because there is no dedicated abstraction for course collisions in the base application. This scenario illustrates one part of our claim that simple declarative role mappings are not sufficient. Binding a CI type is, in general, not a simple equation of it with a type in the base application. It rather might imply the collaboration of several instances of the same or of different base types.

```
class SchedulingGraph binds ColoredGraph {
  class CourseVertex binds Vertex wraps Course{
    Edge[] cachedEdges;
    Edge[] getEdges() {
      if (cachedEdges == null) {
        Vector tc = wrappee.getTeacher().getCourses();
        tc.append(wrappee.getStudentYear().getCourses());
        cachedEdges = new Edge[tc.length];
        for (int i=0;i<tc.length;i++) {
          Course x = (Course) tc[i];
          cachedEdges[i] = CourseCollision(wrappee,x);
        }
      }
      return cachedEdges;
    }
    void setColor(int color) {
      wrappee.timeSlot = TimeSlots.getSlot(color);
    }
  }
  class CourseCollision binds Edge {
    Course c1,c2;
    CourseCollision(Course c1, Course c2) {
      this.c1=c1; this.c2 = c2;
    }
    Vertex getV1() { return CourseVertex(c1); }
    Vertex getV2() { return CourseVertex(c2); }
  }
}
```

Figure 4.20: Binding for scheduling graph

The example at hand also allows us to illustrate how our proposal deals with the requirement for multiple bindings of the same interface to different abstractions in the base application - the other part of our claim that simple declarative role mappings are not sufficient. This was exemplified by the “Student knows Teacher” graph in Tab. 4.1, where both, students and teachers, play the role of vertices in this graph. In the previous section, we indicated that in our model, this can be expressed by implementing different bindings of the same interface with different names, which is now illustrated in Fig. 4.21. Here we have multiple bindings of the interface `Vertex` without, however, introducing ambiguity, because the bindings can still be discriminated by the different names, `StudVertex` and `TeacherVertex`, respectively.

Finally, we would like to use the more sophisticated example to discuss more advanced issues of wrapper recycling. Consider the wrapper recycling calls `CourseCollision(c,x)` in `CourseVertex.getEdges` in Fig. 4.20,

```
class StudentKnowsTeacherGraph binds Graph {  
  
    class StudVertex binds Vertex wraps Student {  
        ...  
    }  
  
    class TeacherVertex binds Vertex wraps Teacher {  
        ...  
    }  
  
    ... Vertex v1 = StudVertex(aStudent); ...  
    ... Vertex v2 = TeacherVertex(aTeacher); ...  
}
```

Figure 4.21: Multiple bindings of the same interface

which ensure that there is only one unique instance of `CourseCollision` for each pair $(c1, c2)$ of courses. In this example, an undirected edge in the course collision graph is represented by two directed edges, therefore a wrapper recycling call `CourseCollision(c1, c2)` will in general yield a different wrapper than `CourseCollision(c2, c1)` – In other words: wrapper recycling takes the order of the constructor arguments into account. If we want to disregard the order of the arguments, this can be done with an appropriate data structure. For example, a direct representation of undirected edges would also be possible if we would pass a *set* with the two courses as elements in the `CourseCollision` constructor calls instead of the ordered pair of courses.

Now, that we have illustrated advanced issues of bindings, let us focus on the reuse supported by our proposal along this dimension. For this purpose, Fig. 4.22 shows a completely different view of the university application as a graph. The classes defined in Fig. 4.22 remodularize the university application to present the student contacts graph. In this graph, students play the role of vertices and two vertices are connected if the students visit a joint course. The class `StudContactsGraph` represents the general remodularization to the `Graph` collaboration, while `StudContactsColoredGraph` and `StudContactsMatchedGraph` refine this class in order to specialize the collaboration to `ColoredGraph` and `MatchedGraph`⁶, respectively. A minimum coloring in the student contacts graph would represent maximum groups of students that do not know each other and would therefore be good candidates for joint exams with little cheating opportunities. A maximum matching, on the other hand, would be helpful to assign the students to two person apartments, such that most students are pooled together with

⁶The code for the `MatchedGraph` CI and its implementation are not shown but are analogous to the coloring example


```
class StudContactsGraph binds Graph {
  class StudVertex binds Vertex wraps Student {
    Edge[] getEdges() {
      ... Student t = ... ;
      ... e[i] = StudContact(wrappee,t);
      return e;
    }
  }
  class StudContact binds Edge {
    Student s,t;
    StudContact(Student s, Student t) {
      this.s = s; this.t = t;
    }
  }
}
class StudContactsColoredGraph extends StudContactsGraph
                                binds ColoredGraph {
  override StudVertex {
    void setColor(int c) {
      Exam.joinGroup(s,c);
    }
  }
}
class StudContactsMatchedGraph extends StudContactsGraph
                                binds MatchedGraph {
  override StudContact {
    void setMatched(boolean b) {
      Rooms.getFreeApartment().assignStudents(s,t);
    }
  }
}
```

Figure 4.22: Alternative bindings of ColoredGraph and MatchedGraph

a person they know.

The sample code in Fig. 4.22 is presented for illustrating two nice features of our model. First, together with the code in Fig. 4.20, it demonstrates how two worlds of types can be multiply mapped to each other without ever being changed. The second feature is again due to the seamless integration of our new concepts into the standard object-oriented concepts of classes, inheritance and subtype polymorphism. Inheritance allows us to reuse **StudContactsGraph** in the definition of both **StudContactsColoredGraph** and **StudContactsMatchedGraph**. Similar to the refinements of nested interfaces in Fig. 4.17, and the refinements of nested implementation classes in Fig. 4.19, the nested bindings **StudVertex** and **StudContact** of **StudContactsGraph** can be refined with an **override** declaration, as illustrated

```

class SucAugSched extends ColoredGraph<
    SuccessiveAugmentationColoring,SchedulingGraph> {}
class SimAnSched extends ColoredGraph<
    SimulatedAnnealingColoring,SchedulingGraph> {};
...
final SchedulingGraph sg =
    wantSucAug ? new SucAugSched() : new SimAnSched();
sg.computeMinimumColoring( sg.CourseVertex[] (courses) );

```

Figure 4.23: Demo code

by `StudContactsColoredGraph.StudVertex` and `StudContactsMatchedGraph.StudContact`.

4.4.4 Polymorphism

As introduced in Sec. 4.3.3, implementations and bindings of a CI can be freely combined. In terms of the graph example, this is illustrated in Fig. 4.23, where two different complete realizations of `ColoredGraph` (cf. Fig. 4.17) are defined by combining the same binding class, `SchedulingGraph`, with two different implementation classes, `SuccessiveAugmentationColoring` and `SimulatedAnnealingColoring`.

Both combinations, `SucAugSched` and `SimAnSched`, are subtypes of their common binding part, `SchedulingGraph`, and can therefore uniformly be used whenever an object of type `SchedulingGraph` is expected. `SucAugSched` and `SimAnSched` are two instantiations of `SchedulingGraph` that differ from each other on the coloring algorithm. This allows us to write code like the `computeMinimumColoring()` call in Fig. 4.23 polymorphically with respect to the coloring algorithm used.

Other examples of this kind, i.e., the same binding classes being combined with different implementation classes can be found in Fig. 4.24. Just as `SchedulingGraph`, `StudContactsColoredGraph` binding from Fig. 4.22 can be combined with any of the `ColoredGraph` implementations presented in Fig. 4.18. Similarly, `StudContactsMatchedGraph` can be combined with an arbitrary matching algorithm that implements `MatchedGraph` (adumbrated in Fig. 4.24). Both `SucAugStudContacts` and `SimAnStudContacts` are subtypes of `StudContactsColoredGraph` and can be used everywhere it is expected. Similarly, `Matching1StudContacts` and `Matching2StudContacts` are subtypes of `StudContactsMatchedGraph`.

On the reverse side, one could think of coloring algorithms, i.e., of implementation types, as being parameterized with the expected facet of their CI. Hence, any operation written to an implementation type, is naturally polymorphic with respect to all bindings of that implementation type's CI. For instance, `SucAugStudContacts` from Fig. 4.24 and `SucAugSched` from

```

class SucAugStudContacts extends ColoredGraph<
    SuccessiveAugmentationColoring, StudContactsColoredGraph> {}
class SimAnStudContacts extends ColoredGraph<
    SimulatedAnnealingColoring, StudContactsColoredGraph> {}
class Matching1StudContacts extends MatchedGraph<
    MatchingAlgorithm1, StudContactsMatchedGraph> {}
class Matching2StudContacts extends MatchedGraph<
    MatchingAlgorithm2, StudContactsMatchedGraph> {}

```

Figure 4.24: Free combination of components and connectors

Fig. 4.23 represent two instantiations of `SuccessiveAugmentationColoring`, i.e., both are subtypes of the latter.

To summarize the typing relationships: If we have a class `C` `extends` `CI<A,B>` with implementation class `A` and binding class `B` which communicate over a common collaboration interface `CI`, then `C` is a subtype of both `A` and `B`, which are subsequently both subtypes of `CI`.

Since `A` and `B` are independent classes, we have to deal with conflicts which are caused by accidental name clashes of methods in `A` and `B`. We resolve these possible conflicts by hiding all methods of `A` and `B`, which are not already in the collaboration interface `CI`, in the context of a reference of type `C`. For example, if both `A` and `B` would introduce a method `m()`, then `C c = ...; c.m();` would be an illegal call, whereas `A a = c; a.m();` would be legal, thereby eliminating all possible ambiguities and conflicts.

To recap, we can create completely different and independent mappings of a base structure (e.g., university administration) to a particular component structure (e.g., graph) and combine them with yet a range of different implementations of the component (e.g., different coloring algorithms). In general, the role or task that base objects play in a particular collaboration is not static but depends on the context within which the collaboration is used, e.g., a course is a vertex in the course collision graph, and the same course is an edge in the “teacher uses room” graph (see Tab. 4.1). This is also an advantage of our model over previous more static approaches.

4.4.5 Section Summary

To summarize the section, we want to recall the different dimensions of polymorphism and reuse that are possible in our approach:

- **Collaboration interface dimension:** A hierarchy of collaboration interfaces can be defined, such as the `Graph` interface which is refined by `ColoredGraph` and `MatchedGraph` (see Fig. 4.17).
- **Component dimension:** Multiple independent implementations of a collaboration interface are possible, such as the different coloring

implementations in Fig. 4.18. Component implementations can reuse other component implementations to implement more specialized collaboration interfaces via inheritance. For example, the communalities of the two coloring algorithms in Fig. 4.18 can be outsourced into a common superclass (Fig. 4.19).

- **Connector dimension:** Multiple independent bindings of a collaboration interface to the same or different applications can co-exist, such as the course collision and the student contacts remodularizations in Fig. 4.20 and 4.22, respectively. Inheritance among connectors, such as in Fig. 4.22, allows to reuse existing remodularization specifications when binding more specialized collaboration interfaces.
- **Bound component dimension:** The bound component is a subtype of both the component and the connector type. Therefore, client code, such as in Fig. 4.23, can be reused with any implementation.

4.5 Future Work: Layered Bindings and Implementations

With the inheritance mechanisms introduced in the last section, it is possible to specify different variants of bindings and CI implementations *incrementally*. Furthermore, it is possible to combine any variant of a binding with any variant of a CI implementation. However, it is not yet possible to create different variants of a binding or different variants of an implementation by combining existing variants by layering them on top of each other.

Suppose we have a CI implementation and binding of colored graphs and a CI implementation and binding of matched graphs, but our graph should be both colored and matched. We could create two independent graphs, one for matching and one for coloring, but we may want these graphs to share the same common graph state or interact via method overriding.

This is basically the same problem that was already the topic of Chap. 3 on delegation layers. Indeed, we think that it would be useful to augment CAESAR with means for layering on both the binding side and the implementation side.

The incorporation of delegation layers into CAESAR would have the following consequences:

- **Reusability:** We can combine different variants of CI bindings and implementations, thereby increasing the reusability of individual ‘layers’.
- **Polymorphism:** We get more dimensions of polymorphism, namely in the combination of layers (the ‘parent’ layer is only known by upper bound) and also in the usage of a combination when combining CI binding and implementation (both are only known by upper bound).

- **Dynamics:** The composition happens at runtime, hence the shape of the compound object can depend on runtime values
- **Sharing:** The same layer instance can be shared among many ‘child’ layers (e.g., we could have a binding `B3` that is also instantiated with parent layer `b1`. Furthermore, any (possibly composed) CI binding or implementation object can participate in an arbitrary number of compositions because every particular composition is represented by its own object.

The reason why this section is marked as ‘future work’ is that the integration of delegation layers with the notions of CIs, bindings, and implementations, entails two problems.:

- Some more work on the type system in order to retain static type-safety is required. Delegation layers are composed at runtime and this is not possible offhand with bindings and implementations because - in the presence of subsumption - it is not easy to ensure statically that the parts to be combined expect and provide the same set of methods, respectively.

Suppose we have a collaboration interface `CI` with a provided method `prov1()` and an expected method `exp1()`. Now let `SubCI` be a subinterface of `CI` which adds the methods `prov2()` (provided) and `exp2()` (expected). If a `SubCI` implementation `x` is now polymorphically assigned to a variable of type `CI`, the information that `x` needs to be combined with a binding that implements `exp2()` is statically no longer available.

- If bindings and implementations are represented as objects that are combined at runtime, special care must be taken such that no messages are sent to objects that are not yet operational, e.g., a binding that is not yet connected to an implementation. This is similar to the problem of instantiating abstract classes in Sec. 2.6.

The easiest solution for the first problem is to drop the subtyping relation between a CI and its extensions. With respect to the example above, this means that the assignment of `x` to a variable of type `CI` is rejected by the compiler because `SubCI` is not a subtype of `CI`. Other, more sophisticated type systems that try to preserve the subtyping relation would also be possible, but the importance of the aforementioned subtyping relation would have to be elaborated more clearly in order to justify such an extension.

In order to cope with incomplete objects that are not yet ready to receive messages, we introduce markers that statically mark an object as incomplete. In more detail, we introduce the *binding type* `#CI` and the *implementation type* `@CI` of a collaboration interface `CI`. `CI` is a subtype of

both `#CI` and `@CI` but not vice versa. No messages can be send to an object whose static type is a binding type or an implementation type (including constructor calls of nested classes). Every incomplete instance of a binding `B` of `CI` has type `#B`, which is a subtype of `#CI`, whereby the `#` marker means that no messages can be send to this object, again. Similarly, the type of an incomplete instance of a `CI` implementation `I` is `@I`.

Suppose we have a collaboration interface `CI`, a `CI` binding `B` and a `CI` implementation `I`. The type of an expression `new B()` is `#B`, which is a subtype of `#CI`. Similarly, the type of `new I()` is `@I`, which is a subtype of `@CI`.

With this mechanism we can be sure that an incomplete object will never receive a method call or a constructor invocation. Hence, we can safely apply dynamic layering via delegation on both bindings and implementations, with the same semantics as described in Chap. 3.

The open point is now: How are a binding and an implementation object combined and hence made operational? One solution would be to make this combination just an ordinary layer combination, i.e., one object, say, the binding object `b` would be the ‘parent’ layer and one would be the ‘child’ layer whose parent is assigned at creation time. In this case, the child layer would be the implementation object `i = new I()`. However, this would destroy the symmetry of the combination (the decision whether `CI` implementation or binding plays the child and parent role, respectively, is arbitrary) and would prevent the ‘child’ layer `i` from being shared (in delegation, a child can only have one unique parent).

We think it makes more sense to represent a particular combination of a binding with an implementation by a dedicated object, thereby preserving the symmetry. Syntactically, this could be expressed as follows:

Suppose we have a collaboration interface `CI`, bindings `B`, `B1`, `B2` and implementations `I`, `I1`, `I2` as in Fig. 4.25. Fig. 4.26 shows how these classes could be used as delegation layers, using the same delegation syntax as in Chap. 3. Of particular interest is the expression `CI<b2,i2>`, which creates a new object that represents the particular combination of `b2` with `i2` (note that both `b2` and `i2` are already composed objects).

To summarize, the incorporation of layering for `CI` bindings and implementations seems to be an interesting new possibility. However, how useful these new features really are has not yet been sufficiently explored. In addition, although the extended type system preserves type safety, it makes the language more complicated. A more lightweight solution would be desirable.

4.6 Related Work

Pluggable Composite Adapters (PCAs) [MSL01] and their predecessor, *Adaptive Plug and Play Components (APPCs)* [ML98], have been important

```
interface CI {
    provided void foo();
    expected void bar();
    interface Nested { ... }
}

class B binds CI {
    void bar() { ... }
    class BNested binds Nested {...}
}
class B1 extends B {
    void bar() {...super.bar();...}
    override class BNested { ... }
}
class B2 extends B {
    void bar() { ...super.bar();...}
    override class BNested { ... }
}

class I implements CI {
    void foo() { ... }
    class Nested {...}
}
class I1 extends I {
    void foo() {...super.foo();...}
    override class Nested { ... }
}
class I2 extends I {
    void foo() { ...super.foo();...}
    override class Nested { ... }
}
```

Figure 4.25: Classes to be used as delegation layers

```
// B1 b1 = new B1(); rejected by compiler
#B1 b1 = new B1();
// b1.bar(); rejected by compiler
#B2 b2 = new B2<b1>(); // combine
@I1 i1 = new I1();
@I2 i2 = new I2<i2>(); // combine
CI<B2,I2> ci = CI<b2,i2>;
B2 b2 = ci; // OK
I2 i2 = ci; // OK
CI ci2 = ci; // OK
```

Figure 4.26: Layer combination of CI bindings and implementations

starting points for this work. Both approaches offer different means for on-demand remodularization. The APPC model had a vague definition of required and provided interfaces. However, this feature was rather ad-hoc and not well integrated with the type system. Recognizing that the specification of the required and expected interfaces of components was rather ad-hoc in APPCs, PCAs even dropped this notion and reduced the declaration of the expected interface to a set of standard abstract methods. With the notion of collaboration interfaces, the approach presented here represents a qualitative improvement over PCA and APPC.

Due to the lack of decoupling of the component implementations from their bindings, the connectors and adapters in APPC and PCA models are bound to a fixed component. Furthermore, the lack of the notion of virtual types is another drawback of these approaches as compared to the work

presented here. In addition, both approaches rely on a dedicated mapping sublanguage that is less powerful than our notion of object-oriented wrappers with wrapper recycling. Among these approaches, the APPC model of remodularization is a class-based one, and only PCAs share the object-based on-demand remodularization with our approach.

In [HM00], a variant of the PCA construct, called *dynamic view connectors (DVCs)* is used in the architecture of an integrated software engineering environment to support the late integration of independently developed software engineering tools. This work demonstrates the power of on-demand remodularization in a real-world, fairly large system. By being basically a realization of the PCA concept, DVCs also share their shortcomings mentioned above.

The Hyperspaces model and its instantiation in Hyper/J [TOHS99] also support on-demand remodularization - this notion was actually first introduced by the Hyperspaces model. Both, on-demand remodularization in Hyper/J and in our approach, have a common goal: “On-demand remodularization allows a developer to choose at any time the best modularization, based on any or all of their concerns, for the development task at hand” [OT00]. Hence the same name.

However, despite the common goal, there are some important differences between these two approaches. In a nutshell, the functionality offered by Hyper/J can be summarized as *extracting concerns* and *composing concerns*.

Extracting concerns means that one can take a piece of existing software and tag parts of the software, e.g., method `a()` in class `A` and method `b()` in class `B`, by means of a so-called *concern mapping*. Later, this mapping can be used to extract a particular concern from this software and reuse it in a different context. This is similar to the old idea of retroactive generalization in inheritance hierarchies [Ped89]. An important concept for extracting concerns is the notion of *declarative completeness*. Basically, this means that all methods that are used inside the tagged methods but are not tagged themselves are declared as *abstract* in the context of the extracted concern. Our model does not have any dedicated means for feature extraction.

However, we think that with respect to *composing concerns* our approach is in some important ways superior to Hyper/J. Composition in Hyper/J happens by means of a so-called *hypermodule specification*, which describes in a declarative sublanguage, how different concerns should be composed. In terms of our model, a hypermodule performs both the functionality of our binding classes and the actual composition with a specific concern implementation. Due to this mixing and due to the absence of an interface concept similar to our collaboration interface, Hyper/J has no polymorphism and reuse as in our approach, e.g., one cannot switch between different implementations and bindings, and one cannot use them polymorphically. Since the mapping sublanguage is declarative, it relies on similar signatures that can be mapped to each other, and transformations other than name

transformations (e.g., type transformations), are very difficult. In addition, Hyper/J's sublanguage for mapping specifications from different hyperslices is fairly complex and not well integrated into the common OO framework.

The last important difference is that Hyper/J's approach is class-based: it is not possible to add the functionality defined in a hyperslice to individual objects, instead the objects have to be created as objects of the compound hypermodule from the very beginning. Therefore, multiple independent bindings that are added to individual objects at runtime are not possible.

At this point, the question rises of how to position the work presented here with respect to previously published works on *collaboration-based decomposition* (CBD). CBD approaches aim at providing modules that encapsulate a whole collaboration of classes. With CBD classes are decomposed into the roles they play in the different collaborations. The idea is nicely visualized by a two dimensional matrix with the classes as the column indexes and collaborations in which these classes are involved as the row indexes.

Mixin Layers [SB98] and delegation layers [Ost02, see also Chap. 3] are two representatives of approaches to CBD. Both approaches provide concepts for composing and decomposing a collaboration into *layers*, such that a particular collaboration variant can be obtained by composing the required layers. Mixin layers use a nested variant of mixin-inheritance [BC90], whereas delegation layers combine delegation and virtual classes in order to defer the layer combination until runtime. None of these approaches support on-demand remodularization. The definition of a collaboration layer in these approaches also encodes how the collaboration will be integrated. The vocabulary of abstractions that are involved in an application is defined a-priori to the definition of any collaboration layer and is consequently shared by all layer definitions.

VanHilst and Notkin propose an approach for modelling collaborations based on templates and mixins as an alternative to using frameworks [VN96]. However, this approach may result in complex parameterizations and scalability problems. A *contract* [Hol92] allows multiple potentially conflicting component customizations to exist in a single application. However, contracts do not allow conflicting customizations to be simultaneously active. Thus it is not possible to allow different instances of a class to follow different collaboration schemes.

Lasagne [TVJ⁺01] is a runtime architecture that features aspect-oriented concepts. An aspect is implemented as a layer of wrappers. Aspects can be composed at run-time, enabling dynamic customization of systems, and context-sensitive selection of aspects is realized, enabling client-specific customization of systems.

Hölzle [Höl93] analyses some problems that occur when combining independent components. Our proposal can be seen as an answer to the problems and challenges discussed in [Höl93]. Mattson et al [MBF99] also indicate the

problems with framework composition, analyze reasons for these problems and investigate the state of the art of available solutions. In [Bos98], Bosch proposes a language construct for specifying a class as the adapter of another class, that is, for explicit expression of the adapter pattern. The adapter construct as proposed in [Bos98] has two main restrictions: First, it does not support adaptation of entire collaborative functionality. Second, as indicated in [Bos98], it does not allow interface incompatibility.

Our work is also related to *architecture description languages* (ADL) [SG96], for example Rapide [LKA⁺95], Darwin [MK96], C2 [MOT97], and Jiazzi [MFH01]. The building blocks of an architectural description are components, connectors, and architectural configurations. A component is a unit of computation or data store, a connector is an architectural building block used to model interactions among components and rules that govern those interactions, and an architectural configuration is a connected graph of components and connectors that describe architectural structure. In comparison with our approach, ADLs are less integrated into the common OO framework, and do not have a dedicated notion of on-demand remodularization in order to provide a new virtual interface to a system.

We think that collaboration interfaces might also prove very useful in the context of ADL. In ADL, components also describe their functionality and dependencies in the form of required and provided methods (so-called *ports*). The goal of these ports is to render the components reusable and independent from other components. However, although the components are syntactically independent, there is a very subtle semantic coupling between the components, because a component A that is to be connected with a component B has to provide the exact counterpart interface of B. The situation becomes even worse if we consider multiple components that refer to the same protocol. The problem is that there is no central specification of the communication protocol to which all components that use this protocol can refer to – in other words: we have no notion of a collaboration interface.

4.7 Chapter Summary

This chapter proposed language concepts that facilitate the separation of an application into independent reusable building blocks and the integration of pre-build generic software components into applications that have been developed by third party vendors.

A key element of our approach is the notion of *collaboration interfaces*, used to declare the type of generic components. Collaboration interfaces are nested interfaces, bundling several abstractions that together build up the concept world of a component type into a family of virtual types [Ern01]. In addition to the ‘client-from-server contract’, expressed by standard interfaces, collaboration interfaces also capture what servers expect from poten-

tial client contexts in which they might be integrated, i.e., the server-from-client contract. The implementations of these two contracts are completely decoupled from each other.

The implementation of the second contract translates the abstractions and vocabulary of an existing code base into the vocabulary understood by a set of components that are connected by a common collaboration interface. This translation is called *on-demand remodularization*, since the translation is virtual and effective only during the execution of functionality in the collaboration interface, whose server-from-client contract is implemented by the remodularization. Our approach to remodularization is object-based and uses the full computational power of an object-oriented language. The concept of *wrapper recycling* was additionally introduced to support the specification of the remodularization.

The decoupling of component implementation from bindings via remodularizations, allows to mix-and-match remodularizations and components on demand. This decoupling combined with the lean integration of collaboration interfaces with generalized notions of inheritance and subtype polymorphism, provide for a high degree of reuse in our model.

Combining Crosscutting Models

This chapter shares material with the paper ‘Conquering Aspects With Caesar’ [MO03] which has been presented at AOSD 2003.

The language which has been presented in the previous chapter shows how different models can be represented and translated to each other. However, so far the combination of two models is purely *additive*. Being additive means that the creation of a model combination has no effect on existing code. This is similar to subclassing: Creating a subclass of a class does not change the semantics of existing code because the existing code will still create instances of the superclass. Usual subtype polymorphism is the only means to change the behavior of existing code.

We will see that this is insufficient for an important class of concerns, namely those that interact with other concerns in a crosscutting way. For this purpose, we will add *pointcuts*, *advices* and *aspect deployment* to CAESAR. Pointcuts describe points in the call-graph of a program, and advices describe actions to be executed at these pointcuts. Pointcuts and advices are new language concepts that are frequently seen as the hallmark of aspect-oriented programming. In this chapter we will argue that they should be combined with our notions of model translation in order to realize modules for crosscutting models. Aspect deployment is a new concept for the polymorphic composition of crosscutting models and for fine-grained control over the effects of pointcuts and advices.

In order to set the scene, we start with a very different yet enlightening problem statement. In the previous chapter, we motivated our model from the perspective of integrating independent components, whereas in this chapter we begin by examining problems with existing aspect-oriented languages that have pointcuts and advices only.

5.1 Introduction

A popular view of aspects is one of modules that define (i) which points in the execution of a base program to intercept and (ii) how to react at these points. This is roughly speaking the definition of aspects in AspectJ - the best-known AO-language. Without questioning the power of join point interception (JPI), we believe, however, that more powerful means for structuring aspect code are needed on top of it. Especially, better support is needed (a) for expressing an aspect as a set of collaborating abstractions, comprising the modular structure of the world as seen by the aspect, and (b) for structuring the interaction between two parts of an aspect: *aspect implementation*, and *aspect binding* (integration) into a particular code base.

To clarify the terminology, let us consider a simple and well-known example: the subject-observer pattern [GHJV95]. The world as seen by this aspect consists of two abstractions, subject and observer, which are mutually recursive in that the definition of each of them refers to the other one. The definition of an aspect should clearly define these two abstractions as two modules that interact with each other via well defined interfaces. This is basically what we refer to in point (a) above.

Now, let us consider the distinction that we make between aspect implementation and aspect binding in point (b) above by the example of the subject-observer protocol. The implementation part comprises in this case the implementation of methods such as `addObserver()`, `removeObserver()` and `changed()`, say by means of a `LinkedList`¹. The binding part, on the other hand, comprises details about how to integrate the observer protocol into a particular context mapping the roles “Subject” and “Observer” to particular application classes, e.g., `JButton` and `MyActionListener`. An example for such binding details would be the extraction of the part of the subject state (e.g., `JButton`) to be passed over to the observers along a change notification, as well as how the notification is performed in terms of the method to call on the observer site.

The advantage of supporting the definition of an aspect as a set of mutually recursive abstractions that interact via well-defined interfaces is more or less a direct derivate of the advantages of the object-oriented approach to modeling a world of discourse; for this reason it does not require particular justification at this stage of the discussion. A short discussion is needed, though, to justify the requirement for decoupling aspect implementation from aspect binding, which we proceed with in the following two paragraphs.

An aspect implementation that is tightly coupled with a particular aspect binding, by the virtue of being defined within the same module, cannot

¹Of course, other implementations are possible, e.g., one that executes the observer notifications asynchronously, or one that employs buffering to eliminate duplicated notifications.

be reused with other possible bindings. Hence, this particular aspect implementation must be rewritten for every meaningful binding, thereby rendering the application tangled because the aspect implementation becomes itself crosscutting (due to code copying). Especially for non-trivial aspects with complex implementations, rewriting of the aspect implementation is tedious and error-prone.

On the other hand, an aspect binding that is tightly coupled to a specific aspect implementation is also undesirable. An aspect binding can be seen as a translator which translates the concepts, terms, and abstractions of the application's world into the world of the particular aspect domain, whose usage is not limited to a specific aspect implementation. Consider for example an aspect binding that transforms a particular business application data model to the domain of graphs with nodes and edges. Such a graph representation is useful in many places, not just for one particular aspect implementation.

Without dedicated language support it is rather difficult to separate aspect implementation and binding properly. We will elaborate on this claim in Sec. 5.2, where we investigate the AspectJ approach to separation of aspect implementation and binding by means of abstract aspects/pointcuts and inheritance. In addition, the discussion in Sec. 5.2 will also reveal the deficiencies of AspectJ's JPI-based approach with respect to modeling multiple mutually recursive abstractions.

As a response to the problems we identify, we propose the CAESAR model, which is based on the notion of collaboration interfaces (CI) as presented in Chap. 4 as a means to better support a-posteriori integration of independent components into existing applications. In this chapter we show that CIs and the related notions of separated CI implementations and CI bindings, once properly adopted to the needs of aspect-orientation, can also be applied to support a more modular structuring of aspect code and better aspect reuse. This is because in CAESAR object-oriented concepts for flexible composition, such as subtyping and polymorphism, which proved so useful for classes and objects, naturally apply to aspects, too.

We have picked up an AspectJ example as the object of our investigation for the simple reason that AspectJ is the most prominent approach that emphasizes the structuring of aspects around join points and advices. There are other approaches, including HyperJ, [TOHS99], that have some similarity to ours in their goals, but that have a very different technical realization. We will discuss these approaches in their relation to CAESAR in the related work section.

The reminder of this chapter is organized as follows. Sec. 5.2 sets the scene by discussing problems with existing aspect-oriented languages. After being presented in Sec. 5.3, the CAESAR model will be evaluated in Sec. 5.4 with respect to the problems identified in Sec. 5.2. Sec. 5.5 discusses related work. Finally, Sec. 5.6 summarizes the chapter and outlines future work.

```
public abstract aspect ObserverProtocol {
    protected interface Subject { }
    protected interface Observer { }
    private WeakHashMap perSubjectObservers;
    protected List getObservers(Subject s) {
        if (perSubjectObservers == null)
            perSubjectObservers = new WeakHashMap();
        List observers =
            (List) perSubjectObservers.get(s);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(s, observers);
        }
        return observers;
    }
    public void addObserver(Subject s, Observer o){
        getObservers(s).add(o);
    }
    public void removeObserver(Subject s, Observer o){
        getObservers(s).remove(o);
    }
    abstract protected void
        updateObserver(Subject s, Observer o);

    abstract protected pointcut subjectChange(Subject s);

    after(Subject s): subjectChange(s) {
        Iterator iter = getObservers(s).iterator();
        while ( iter.hasNext() )
            updateObserver(s, ((Observer)iter.next()));
    }
}
```

Figure 5.1: Reusable observer protocol in AspectJ

5.2 Problem statement

In this section we discuss the deficiencies of a JPI-based approach to aspect structuring. Please note that the discussion in this section is by no way a critique on the notions of JPIs and advices. On the contrary, recognizing them as pivotal concepts of aspect-oriented languages, we emphasize the need for higher-level module concepts on top of them.

For illustrating the problems, we use as an example the implementation of the observer pattern in AspectJ proposed in [HK02] by Hannemann and Kiczales, as shown in Fig. 5.1 and Fig. 5.2, whereby Fig. 5.1 shows a reusable implementation of the observer protocol in AspectJ, while Fig. 5.2 binds it to particular classes.

```

public aspect ColorObserver extends ObserverProtocol
    declare parents: Point implements Subject;
    declare parents: Line implements Subject;
    declare parents: Screen implements Observer;

    protected pointcut subjectChange(Subject s):
        (call(void Point.setColor(Color)) ||
         call(void Line.setColor(Color)) ) && target(s);

    protected void updateObserver(Subject s, Observer o) {
        ((Screen)o).display("Color change.");
    }
}

```

Figure 5.2: Binding of observer protocol in AspectJ

The basic idea in Fig. 5.1 is that the aspect `ObserverProtocol` declares an *abstract pointcut* that represents change events in the `Subject` classes. The empty interfaces `Subject` and `Observer` are marker interfaces that are used in the binding to map the application classes to their roles. The observers for each subject are stored in a global `WeakHashMap` (the weak references are required in order to prevent a memory leak) that maps a subject to a list of observers. In case of a subject change all observers are notified by means of the abstract method `updateObserver()`, which is overridden in the binding aspect in order to fill in the appropriate update logic.

This proposal has two main advantages. First, Fig. 5.1 is indeed a reusable implementation of the observer protocol: Nothing in the implementation is specific to a particular binding of this functionality. This is because the authors [HK02] recognize the need to separate aspect implementation and aspect binding. Second, the same role, e.g., `Subject`, can be mapped to multiple different classes, e.g., `Point` and `Line` as in Fig. 5.2. It would also be no problem to assign two roles, e.g., `Subject` and `Observer`, to the same class, or assign the same role twice to the same class in two different bindings. For example, a `Point` can be simultaneously a subject concerning coordinate changes as well as color changes. In terms of [Szy96], the observer “component” in Fig. 5.1 is independently extensible.

These features are probably the rationale for the author’s decision against an alternative (simpler) implementation of the observer protocol in AspectJ. The alternative solution of which we speak is to declare `addObserver()` and `removeObserver()` in the interface `Subject` and then (in the binding) inject these methods into the corresponding classes by means of a so-called *introduction*, - AspectJ’s open class mechanism. Similarly, a `LinkedList` could be introduced into every `Subject` class, thereby rendering the `perSubjec-`

`tObservers` map unnecessary. However, with this solution, a class could not have two different instantiations of the `Subject` role, because then the class would have multiple implementations of the same method (e.g., `addObserver()`), hence resulting in a compiler error. In other words, we would lose independent extensibility.

Now, let us take a critical look on this solution. We identify the following problems.

Lacking support for multi-abstraction aspects

Note that all methods in Fig. 5.1 and 5.2 are top-level methods of the enclosing aspect class. For example, `addObserver()`, which is conceptually a method of the subject role, is a top-level method whose first parameter is the respective `Subject` object. This design decision is conceptually questionable because the methods of all aspect roles are forced into a flat list of methods. In a way, this is a rather procedural style of programming, contradictory to one of the fundamentals of object-oriented programming, according to which a type definition contains all methods that belong to its interface. It is also contradictory to the aspect-oriented vision of defining crosscutting modules in terms of their own modular structure. The structure of the aspect in Fig. 5.1 is one of empty abstractions and unstructured method definitions, and as such not particularly modular.

The implications of this design decision are not only of a conceptual, but also of a practical nature. First, we cannot pass objects that play a role `R` to other classes that expect an instance of that role. Envisage, for illustration, a role `Comparable` with a method `compareTo()`. If we want to pass an object as a `Comparable` to another class, e.g., a sorting class, then the approach in Fig. 5.1 and 5.2 based on introducing an empty interface and encoding all methods as top-level methods of the enclosing class, does not work. The alternative would be to use AspectJ's introduction mechanism to introduce the interface and its methods directly into the respective class but then again we would lose independent extensibility, as discussed above. For example, a `Point` could be compared to another `Point` by means of their geometrical distance $\sqrt{x^2 + y^2}$ as well as their Manhattan distance $\|x\| + \|y\|$ to the origin, which would require two independent implementations of the `Comparable` abstraction.

A similar problem shows up, if some interaction between the abstractions that build up the aspect's model of the world - `Subject` and `Observer` in our example - is needed. The interaction in Fig. 5.1 is very simple: a subject passes itself on calling the `notify` method on each observer, but the parameter gets never used in the binding of the aspect in Fig. 5.2. It is more realistic that observers would want more detailed information of what state change actually happened on the subject's site. This would require some query methods in the interface of the subject. Using the AspectJ

design "pattern" exemplified in Fig. 5.1 and Fig. 5.2, where abstractions are typeless, we would have to declare such query methods also at the top level, e.g., `getState(Subject s)`. The query methods would have to be declared abstract in Fig. 5.1 since their implementation is binding specific and should be implemented by the concrete binding subaspect in Fig. 5.2. However, it is not possible to implement different query methods for `Point` and `Line`, i.e., it is not possible to dynamically dispatch with regard to the type of the base objects being decorated with the subject functionality.

Another practical issue related to having only top-level methods and empty abstraction is that the late binding dispatch happens only with regard to the instance of the enclosing class (since the methods are methods of the enclosing class). This means that it is not possible to assign different implementations of these methods for different subroles of the aspect. Consider for example the case that we have a specialization `SpecialSubject` of `Subject` that requires a different implementation of `addObserver()`, e.g., one that checks for double registration of observers. We would then introduce an additional `addObserver()` method whose first parameter has type `SpecialSubject`, but since the dispatch is only based on the class of the receiver object (and not of the argument classes) subsumption for instances of `SpecialSubject` does not work as expected, i.e., if a `SpecialSubject` would be polymorphically assigned to a variable of type `Subject`, a subsequent call to `addObserver()` would execute the wrong method.

With the solution in Fig. 5.1 and 5.2 it is also pretty awkward to associate state with the individual abstractions in the definition of the aspect. For example, the observers of all subjects are stored in a global hash map `perSubjectObservers`. Besides the dangers of such a global bottleneck, the access and management of state becomes pretty clumsy. The example in Fig. 5.1 is relatively simple because state is associated with only one of the abstractions (`Subject`) and this state consists of only one field. However, the general case is that multiple abstractions in the module structure of the aspect may declare multiple fields. A simple example would be an implementation where observers maintain a history of the observed state change, e.g., when they need to react on change bundles rather than on individual changes. If we consider the case that all roles need many different fields then the code might very easily become a mess, if all these fields are hosted by the outer aspect.

The problem with modeling state becomes even worse, once we consider the case of role inheritance, e.g., `SpecialSubject` inheriting from `Subject`. In this case, we would end up simulating shared data fields manually. This problem with modeling state applies to the aspect binding as well. There we might also want to associate state with the objects that are mapped to the aspect roles, e.g., in order to cache computed values.

Summarizing the problems so far, what we would like to have is a nested class structure of aspect implementation and aspect binding within which

we can assign methods and state to every aspect role in isolation.

Lacking support for sophisticated mapping

The second kind of problem with the solution in Fig. 5.1 and 5.2 is that the mapping from aspect abstractions to base classes by means of the `declare parents` construct works only when each aspect abstraction has a corresponding base class to which it is mapped directly. However, this is not always the case. Consider e.g., a scenario in which there is no class `Line` and every `Point` object has a collection of neighbor points. If we want to map this data structure to a graph aspect defined in terms of `Node` and `Edge` abstractions, then an edge would be represented by two adjacent points, but there is no abstraction in the base application to which we can map the `Edge` abstraction. The latter is only implicitly and indirectly represented by the collections of adjacent points.

Lacking support for reusable aspect bindings

Third, every aspect binding is coupled to one particular implementation. For example, the `ColorObserver` binding in Fig. 5.2 is hardwired to the observer pattern implementation in Fig. 5.1, although the binding itself is not dependent on the implementation details of the observer pattern. The observer pattern is not a very good example to illustrate the usefulness of a binding that can be used with many different implementations; a better example is that of an aspect binding that maps an arbitrary data structure, e.g., the classes of an abstract syntax tree, to a general tree representation. Many different implementations of a tree make sense in conjunction with such a binding, e.g., one that displays trees on the screen or one that performs algorithms on trees. That is, one might want to be able to write some functionality that is parameterized with a particular binding type, but is polymorphic with respect to the implementation. This is, however, not possible, if the binding is coupled to the implementation.

Lacking support for aspectual polymorphism

The fourth deficiency concerns aspect deployment. We say that the `ColorObserver` aspect in Fig. 5.2 is statically deployed. By this we mean that once compiled together with the package containing the figure classes, the changes in the particular points in the execution of point and line objects implied by `ColorObserver` aspect are effective. Which is to say that it is not possible to determine at runtime, whether to apply the aspect at all, or which implementation of the aspect to apply, e.g., a `LinkedList` version, or one with asynchronous notifications. We say that *aspectual polymorphism* is missing, in the sense that the code is not polymorphic with respect to the types and implementations of the aspects affecting it after compilation.

As an analogy, let us consider how subclass polymorphism is used to separate concerns that relate to different kinds of a data abstraction in object-oriented languages. Given e.g., a class `Order` (think: aspect) and a class `OrderProcessing` that operates on orders (think: base code²), and the requirement to introduce a special kind of order, say, `ExpressOrder`, we do not need to "weave" `OrderProcessing` either with `Order` or `ExpressOrder`. Due to subtype polymorphism of object-oriented languages, any abstraction that we write – also our `OrderProcessing` class – has the built-in potential of being incrementally extended with future variations of the objects on which it operates. We can use the functionality offered by `OrderProcessing` polymorphically with both orders and express orders, based on which of these variants we decided to create at runtime. This is not true for the observer base application in our example and potential aspects to be defined in the future, which is why we say that this application lacks aspectual polymorphism.

5.3 Deploying Aspects With Caesar

The basic notions of CAESAR have already been introduced in the previous chapter. In order to model the observer example, we need an appropriate collaboration interface (Fig. 5.3), a sample implementation of the interface (Fig. 5.4), and a binding that maps the observer abstractions to the respective application classes (Fig. 5.5). In the context of this chapter, we will refer to the functionality described by a CI as an *aspect*, to implementations of a CI as *aspect implementations* and to bindings of a CI as *aspect bindings*.

The observer CI in Fig. 5.3 specifies that any implementation of `ObserverProtocol` must provide an implementation of the three provided methods of `Subject`³. On the other side, the expected facet of an aspect makes explicit what the aspect expects from the context in which it will be applied, in order to be able to supply what the provided facet promises. Hence, the expected facet declares methods that are part of the aspect functionality, but whose implementation is binding specific. Consider for instance, the part of the observer protocol concerned with communicating relevant state from the subject to observers, when a change is notified. What part of subject's state is relevant, and how this state should be extracted for being passed to observers is highly dependent on what classes in a particular context play the roles of the subject and observer respectively. Furthermore,

²The analogy may look strange at first because `OrderProcessing` *knows about* `Order` whereas an aspect is not necessarily not known to the base code, but on the other hand this inversion of knowledge is one of the key properties that accounts for the improvement over pure OO

³In this example, the `Observer` abstraction does not happen to have any provided methods. However, one can easily think of other examples where multiple or all abstractions declare a non-empty provided facet.

```
interface ObserverProtocol {
    interface Subject {
        provided void addObserver(Observer o);
        provided void removeObserver(Observer o);
        provided void changed();
        expected String getState();
    }
    interface Observer {
        expected void notify(Subject s);
    }
}
```

Figure 5.3: CI for observer protocol

```
class ObserverProtocolImpl implements ObserverProtocol {
    class Subject {
        List observers = new LinkedList();
        void addObserver(Observer o) { observers.add(o); }
        void removeObserver(Observer o) {
            observers.remove(o);
        }
        void changed() {
            Iterator it = observers.iterator();
            while ( it.hasNext() )
                ((Observer)iter.next()).notify(this);
        }
    }
}
```

Figure 5.4: Sample impl. of observer protocol

the operation to be called on the observer as part of the notification is also binding-specific. This is why `notify()` and `getState()` are declared with the modifier `expected` in Fig. 5.3.

Recall that an implementation of a CI must implement all methods in the provided facet of the CI, i.e., all aspect level `provided` methods, as well as provided facets of all nested CIs. Fig. 5.4 shows a simple implementation of the `ObserverProtocol` CI. Similarly, we could write another implementation of `ObserverProtocol`, say, a class `AsyncObserverImpl` that implements `ObserverProtocol` and realizes a notification strategy with asynchronous updates.

The responsibility of an aspect binding is to implement all expected methods in the aspect's CI and in its nested interfaces. Fig. 5.5 shows a binding of `ObserverProtocol` which maps the subject role to the base classes `Point` and `Line` and the observer role to `Screen`. Every binding has

```
class ColorObserver binds ObserverProtocol {
  class PointSubject binds Subject wraps Point {
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
  }
  class LineSubject binds Subject wraps Line {
    String getState() {
      return "Line colored "+wrappee.getColor();
    }
  }
  class ScreenObserver binds Observer wraps Screen {
    void notify(Subject s) {
      wrappee.display("Color changed: "+s.getState());
    }
  }
}
```

Figure 5.5: (Incomplete) sample binding of observer protocol

```
class C0 extends
  ObserverProtocol<ColorObserver,ObserverProtocolImpl> {};
```

Figure 5.6: Weavelet composition

to provide implementations of all **expected** methods in the corresponding interface, e.g., **ScreenObserver** implements the **notify()** method.

In order to gain a complete realization of a CI, an implementation-binding pair needs to be composed. In the context of this chapter, we call a particular pair of implementation and binding a *weavelet*. An example of a weavelet is the class **C0** which is defined as in Fig. 5.6. This class represents a realization of the **ObserverProtocol** interface that combines the implementation **ObserverProtocolImpl** with the binding **ColorObserver**, as described in Sec. 4.3.3.

5.3.1 Pointcuts and Advices

So far, we have only used the language means already introduced in the previous chapter. However, the code in Fig. 5.3), 5.4), and 5.5 does not specify, how the observer component should be *deployed*. With *deployment* we mean: The creation of instances of the observer component, the definition which instances should be active in which part of the program, and the specification of events that should trigger change notifications.

The last point (specification of events) is done via *advices* and *pointcuts*, similar to those already used in the AspectJ example (Sec. 5.2). The other

```
class ColorObserver binds ObserverProtocol {
  class PointSubject binds Subject wraps Point {
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
  }
  class LineSubject binds Subject wraps Line {
    String getState() {
      return "Line colored "+wrappee.getColor();
    }
  }
  class ScreenObserver binds Observer wraps Screen {
    void notify(Subject s) {
      wrappee.display("Color changed: "+s.getState());
    }
  }
  after(Point p): (call(void p.setColor(Color)))
  {
    PointSubject(p).changed();
  }
  after(Line l): (call(void l.setColor(Color)))
  {
    LineSubject(l).changed();
  }
}
```

Figure 5.7: Binding of observer protocol using pointcuts and advices

deployment-related language means will be discussed in Sec. 5.3.2.

In Caesar, we apply pointcuts and advices primarily in bindings, although they could be applied in any class. Fig. 5.7 shows an **ObserverProtocol** binding that uses pointcuts and advices in order to specify the events that should result in a change notification.

Advices and pointcuts in our model differ from the AspectJ model in two points: One difference concerns the decoration of target objects of a join point with aspect types. This decoration happens implicitly in AspectJ, as shown in the pointcut **subjectChange** in Fig. 5.2, where the base object, **s**, brought into the scope of the aspect **ColorObserver** by the join point **target**, which in this case is either **Line** or **Point**, is automatically seen as being of type **Subject** (see the parameter type of the pointcut).

On the contrary, the conversion happens explicitly in CAESAR, via wrapper recycling calls. In Fig. 5.5, we avoided type conversions in a pointcut, by defining different pointcuts for **Point** and **Line**. This was in order to avoid mingling the discussion on wrapper recycling with that on pointcuts and advices. A shorter variant of the same binding is given in Fig. 5.8. Note the explicit call to wrapper recycling operators, in order to decorate basis

```
class ColorObserver binds ObserverProtocol {  
    ... as before ...  
    after(Subject s):  
        ( call(void Point.setColor(Color))  
          with s = PointSubject(target)) ||  
        ( call(void Line.setColor(Color))  
          with s = LineSubject(target) ) {  
        s.changed();  
    }  
}
```

Figure 5.8: Alternative binding of observer

objects with the aspect facets by means of a **with** clauses, which allows us to bind variables of the pointcut differently in each case of the pointcut. We prefer the explicit variant because it increases programmer’s expressiveness, in that he/she can choose among several constructors of the binding classes, if more than one is available.

However, for the common case that the pointcuts refer to the wrapped objects themselves, we add some syntactic sugar in the form of *embedded advices* (Fig. 5.9). An embedded advice is only possible if the nested class wraps exactly one application object and can be denoted with `wrappee.methodSig`. We deliberately made the syntax similar to the object-oriented notion of method overriding because we see embedded advices as an extended variant of method overriding. The **around** invocations of the original method are similar to a **super** call in usual overriding methods. With usual method overriding, overriding is only active if the object that is the receiver of the respective method call has been explicitly created as an instance of the subclass that overrides the method. With embedded advices, we relax this restriction by allowing to override methods of any object without changing the respective constructor invocations to refer to a subclass of the original class.

The second and more important difference between CAESAR and AspectJ pointcuts and advices is at the semantic level. Compiling a binding class that contains advice definitions does not have any effect on the base application’s semantics. This is because an aspect (its implementation and binding) must be explicitly deployed in CAESAR. Only the advice definitions of explicitly deployed aspects are executed, as elaborated in the following.

5.3.2 Static and Dynamic Deployment

A weavelet has to be *deployed* in order to activate the pointcuts and advices in its binding (recall that pointcuts and advices have no computational effect until they are activated by a deployment). A weavelet deployment is


```
class ColorObserver binds ObserverProtocol {
  class PointSubject binds Subject wraps Point {
    String getState() {
      return "Point colored "+wrappee.getColor();
    }
    void wrappee.setColor(Color c) {
      proceed(c); changed(); }
  }
// embedded advice is equivalent to the
// following code:
// around(Point p): (call(void p.setColor(Color))) {
//   proceed(); PointSubject(p).changed();
// }
}
```

Figure 5.9: Embedded advices

syntactically denoted by the modifier **deploy** and comprises basically two steps: (a) create an instance of the weavelet at hand and (b) call the **deploy** operation on it. One can choose between *static* (load-time) and *dynamic* deployment.

Static deployment

Syntactically, static deployment is expressed by having the **deploy** keyword be a modifier of a **final static** field declaration. Semantically, it means that the advices and pointcuts in the instance that has been assigned to the field become active. For example, the declaration

```
class Test ... {
  deploy public static final CO co = new CO();
  ...
}
```

in an arbitrary class **Test** deploys the instance **new CO()** at load-time. Of course, the object assigned to **co** could also be computed in a **static** method such that the weavelet that is actually deployed might also be a subtype of **CO**, thereby enabling static aspectual polymorphism. The **deploy** keyword can also be used as a class modifier. This variant should be regarded syntactic sugar in the sense that the declaration

```
deploy class CO ... { ... }
```

is equivalent to

```
class CO ... {
```

```
deploy class C0 extends
    ObserverProtocol<ColorObserver, ObserverProtocolImpl>{};
...
void register(Point p, Screen s) {
    C0.THIS.PointSubject(p).addObserver(
        C0.THIS.ScreenObserver(s));
}
```

Figure 5.10: Static Aspect Deployment

```
    deploy public static final C0 THIS = new C0();
    ...
}
```

that is, it is equivalent to declaring a deployed field with name `THIS` as above. Fig. 5.10 shows the full declaration of a statically deployed color observer protocol together with sample code inside `register()` which shows how the deployed weavelet instance can be accessed. Since the instance `C0.THIS` is deployed, the pointcuts inside the aspect declaration are active, i.e., color changes in points and lines will be propagated to `C0.THIS`.

Using `deploy` as a class modifier is appropriate if we need only one instance of the aspect and if aspectual polymorphism is not required. By means of `deploy` as a field modifier we can create and deploy multiple instances of the same weavelet and select from different weavelets using aspectual polymorphism. Having two instances of, say, the `C0` weavelet in the observer example would mean that every `Point` and `Line` would have two independent facets as subject with independent lists of observers. An example that makes more sense is the association of a color or weight to elements of a data structure which can be seen as nodes of a graph. Multiple independent instances of the corresponding weavelet would represent multiple independent colorings of the graph. Other examples can be derived from role modeling, where frequently one object has to play the same role twice, for example, a person is employee in two independent companies. Static aspectual polymorphism is useful if we want to select a particular weavelet based on conditions that are known at load-time. For example, based on the number of processors or the multi-threading support, one might either choose a usual observer pattern implementation or one with asynchronous updates.

Dynamic Deployment

Dynamic deployment is syntactically denoted by the keyword `deploy` be used of as a block statement. The rationale behind dynamic deployment is that frequently we cannot determine which variant of an aspect should be

```
class Logging {
    after(): (call(void Point.setX(int)) ||
        call(void Point.setY(int)) ) {
        System.out.println("Coordinates changed");
    }
}
class VerboseLogging extends Logging {
    after(): (call(void Point.setColor(Color)) {
        System.out.println("Color changed");
    }
}
class Main {
    public static void main(String args[]) {
        Logging l = null;
        Point p[] = createSamplePoints();
        if (args[0].equals("-log"))
            l = new Logging();
        else if (args[0].equals("-verbose"))
            l = new VerboseLogging();
        deploy (l) {
            modify(p);
        }
    }
    public static void modify(Point p[]) {
        p[3].setX(5);
        p[2].setColor(Color.RED);
    }
}
```

Figure 5.11: Polymorphic aspect deployment

applied (or whether we need the aspect at all) until runtime.

Consider for example that we have a program with different logging options, i.e., without logging, with standard logging, and with verbose logging. In CAESAR, this can be implemented as in Fig. 5.11: We have two different logging aspects (`Logging` and `VerBoseLogging`) that are related by inheritance, and we choose among one of these at runtime, depending on the command line arguments with which the program has been started.

The interesting point now is the `deploy` block statement in the main method of `Main`. The meaning of the `deploy` clause is that the advices that are defined in the annotated object `l` become active in the control flow of the `deploy` block, in this case, during the execution of `modify(f)`. In particular, other independent threads that execute the same code would not be affected by the `deploy` clause.

Please note that the advices and pointcuts that will be activated in the `deploy` block are not statically known - for example, `l` is only known by its

```
deploy class LoggingDeployment {
    around(final String s[]): cflow(Main.main(String[]))
        && args(s) && (call(void Main.modify(Point[])) {
        Logging l = null;
        if (args[0].equals("-log"))
            l = new Logging();
        else if (args[0].equals("-verbose"))
            l = new VerboseLogging();
        deploy (l) in { proceed(args); }
    }
}

class Main {
    public static void main(String args[]) {
        Point p[] = createSamplePoints();
        modify(p);
    }
    public static void modify(Point p[]) {...}
}
```

Figure 5.12: Aspect deployment aspects

upper bound `Logging` (`l` could have also been passed as a parameter). In other words, the advices are late bound, similarly to late method binding, hence our term *aspectual polymorphism*. In case `l` is `null` the `deploy` clause has no effects at all.

The usefulness of dynamic deployment becomes clear if we consider a simulation of this functionality by means of static deployment. With static deployment, we would have to encode the different variants by `if` and `case` statements that depend on the current value of switch variables, e.g., statements of the form `if (verboseLogging) {...}`. The structure of the aspect itself would get awkward very quickly because all variants of the aspect are tangled inside the aspect code. In a way, this is similar to simulating late binding in a non-OO language, hence we see dynamic aspectual polymorphism as an imperative consequence of integrating aspects into the OO concept world. Also, such programs would be very fragile with respect to concurrent programs and additional synchronization measures would be required.

An interesting question is whether the aspect deployment code should also be separated from the rest of the code. If desired this can easily be done with another aspect whose responsibility is the deployment of the logging aspect, as indicated in Fig. 5.12. In this figure, the aspect `LoggingDeployment` (which is itself deployed statically) computes and deploys an appropriate logging aspect by means of an `around` advice, i.e., the `proceed()` call is executed in the context of the logging aspect.

5.3.3 Dynamic Deployment and Concurrency

Most other approaches to a more dynamic form of aspects, for example, HandiWrap [BH02], JAC [PSDF01], PROSE [PGA02] or Lämmels formal model [Läm02] enable and disable aspects *globally*, that is, the aspect is simultaneously enabled for all executing threads. At first, our approach of deploying aspects in the context of a particular branch of control flow may seem more complicated, hence we feel that some justification is in place.

One of the motivations for not using global modifications is the fact that it is very hard or even impossible to reconcile global modifications with multithreading. Any thread that is in the middle of something while a new aspect is globally enabled or disabled may become inconsistent. For example, suppose that an aspect inserts an ‘open door’ advice at a pointcut A and a ‘close door’ advice at a pointcut B. If a thread is executing the code that should be guarded by the ‘open/close door’ mechanism while the aspect is enabled in another concurrently running thread, the door will be closed before it was opened. Until now, no general solution to cope with such problems automatically has been proposed (a lot of partly solutions have been proposed, however, see [MPG⁺00] for an overview), and we believe that a general solution might even be impossible.

Hence, our solution does not rely on global modification. In contrast, our solution is thread-safe by construction because only the *current* thread is affected. If the modification should be visible in other threads, too, the programmer can choose the places where it is safe for a thread to add a particular aspect instance to his list of active aspects and add the aspects at these places by means of a `deploy` statement.

5.4 Evaluation

This section evaluates CAESAR with respect to how it copes with the problems outlined in Sec. 5.2. In addition, we will also elaborate on how CAESAR’s explicit aspect instantiation and deployment relate to AspectJ-like languages, where aspects are only implicitly created and which do not have a notion of aspect deployment.

Problems Revisited

This discussion is organized around the five problems identified in Sec. 5.2, as summarized below:

1. Lacking support for multi-abstraction aspects.
2. Lacking support for defining state for individual abstractions pertaining in the definition of an aspect.

3. Lacking support for sophisticated mapping of aspect abstractions to base classes.
4. Lacking support for reuse of aspect bindings with different aspect implementations
5. Lacking support for aspectual polymorphism.

In the following we will explain how each of these problems is solved in CAESAR.

Ad 1: As shown in the code in Fig. 5.3, 5.4, and 5.5, each abstraction in the vocabulary of the world as it is decomposed from the point of view of an aspect, is defined in its own full-fledged module with a well-defined interface. Methods in the interface of one abstraction can be called by methods of other abstractions within the same aspect, or from the outside. Consider e.g., the call of `Subject.notify(...)` in the implementation of `ObserverProtocolImpl` in Fig. 5.4, or the invocation of `CO.THIS.addObserver(...)` in Fig. 5.10.

Due to this finer-grained modularization of the aspect itself, the runtime system is able to dispatch methods not only based on the instance of the aspect, but also based on the particular abstraction in execution. Consider, for example, the `getState()` method in the definition of `Subject`, which was implemented differently for point-subjects and for line-subjects, while being uniformly used in the update logic (cf. Fig. 5.5). As pointed out in Sec. 5.2, the same polymorphism would not be possible, if there were only aspect-level methods. Furthermore, due to the incorporation of virtual classes, it is easy to encode different variants of a multi-abstraction aspect, as exemplified in Fig. 5.13, which shows a variant of the `ColorObserver` binding. Uniformly, we could encode different variants of CI *bindings*.

Ad 2: Let us now consider the issue of defining state for the individual abstractions pertaining to an aspect. As it is shown by examples in the previous section, each abstraction in the modular structure can declare its own state, e.g., `observers` in `Subject`. Hence, there is no need for defining data structures that "globally" maintain aspect-related state of all base objects in a single place, as e.g., `perSubjectObservers` in Fig. 5.1. Similarly, state can be added to the abstractions at the binding side, such as e.g., the `count` field in Fig. 5.13. Furthermore, if inheritance among the participating abstractions is involved, the implied data structure sharing works as expected and is taken care by the runtime system. As pointed out in Sec 5.2, with an approach, where all state is defined globally at the aspect level, the same data structure sharing between inheriting classes would have to be manually simulated in the implementation of the aspect.

```
public class CountingColorObserver extends ColorObserver {  
    override class ScreenObserver {  
        int count = 0;  
        void notify(Subject s) {  
            count++;  
            wrappee.display("Color changed "+count+"times");  
        }  
    }  
}
```

Figure 5.13: Encoding of counting observer using inheritance

Ad 3: In our model bindings are Java classes with some additional features. As such, the definition of mappings from aspect abstractions to the classes of a base application can make use of the full expressiveness of an general purpose object-oriented language. There is nothing to prevent a CAESAR programmer in coding any mapping no matter how sophisticated. A more detailed discussion on this issue supported also by better examples has already been presented in the previous chapter.

Ad 4: Different weavelets can combine an aspect binding with different aspect implementations. On the other hand, different weavelets can combine (and reuse) a particular aspect implementation with multiple different bindings. For example, we can combine the observer protocol binding to `JButton` and `MyActionListener` with the `LinkedList` or the `AsynchronousUpdate` observer implementation, and on the other hand combine the same observer implementation, say `AsynchronousUpdate`, with multiple different bindings, e.g., to `JButton/MyActionListener` and `ListModel/JList`. As a consequence, one can define functionality that is polymorphic with respect to (a) aspect implementations by being written to a certain aspect binding type, (b) aspect bindings by being written to a certain aspect implementation type, or (c) both of them, by being written to a common CI.

Ad 5: As already discussed in Sec. 5.3.2, our approach does support aspectual polymorphism. For example, the `modify(Point p[])` method in Fig. 5.11 is polymorphic with respect to aspects that might be defined in the future. It is even possible to run the same method concurrently within two different threads with and without the logging aspect.

Explicit vs. Implicit Aspect Instantiation/Deployment

Recall that the question we pose here is: How does our notion of explicit aspect instantiation and deployment relate to AspectJ-like languages, within

<pre> aspect A isSingleton { State s; } </pre>	<pre> deploy class A { State s; } </pre>
<pre> aspect A perThis(pointChanges) { pointcut pointChanges(): calls (Point.setX(int)); State s; after(Point p): pointChanges() && this(p) { ...s... } } </pre>	<pre> deploy class A { class PointWrapper wraps Point { State s; } after(Point p): calls(Point.setX(int)) && this(p) { ...PointWrapper(p).s;... } } </pre>
<pre> aspect A perCflow(pointChanges) { pointcut pointChanges(): calls (Point.setX(int)); State s; after(): somePointCut() { ... } } </pre>	<pre> class A { State s; after(): somePointCut {} } deploy class ADepl { around(): calls (Point.setX(int)) { deploy (new A()) { proceed(); } } } </pre>

Table 5.1: Aspect Instantiation in AspectJ (left) and Caesar (right)

which aspects are only implicitly created and which do not have a notion of aspect deployment? In AspectJ, aspect instantiation can be controlled by means of the aspect modifiers `isSingleton` (this is the default), `perThis/perTarget`, and `perCflow/perCflowbelow`. In CAESAR, these aspect instantiation strategies turn out to be special cases or patterns of the more general model in CAESAR.

Tab. 5.1 describes how the AspectJ instantiation strategies can be simulated in CAESAR. The `isSingleton` case is obvious. The `perThis` modifier can be simulated by creating a wrapper class and using wrapper recycling in order to refer to the state that is associated with each point. Simulating `perTarget` is identical to `perThis`, except that we would have to exchange `this(p)` by `target(p)`. More interesting is AspectJ's `perCflow` modifier, which means that an instance of the aspect is created for each flow of control of the join points picked out by the annotated pointcut. The semantics of `perCflow` can be simulated by using a deployment aspect `ADepl` that uses dynamic deployment at the respective starts of control flow.

What do we gain if all the cases in Tab. 5.1 can already be handled very well by AspectJ? To answer this question recall that AspectJ instantiation strategies are just special cases of a more general model in CAESAR. This has two implications. First, we do not need special new keywords to express the semantics of AspectJ instantiation, thereby rendering the conceptual model more slender. Second and more importantly, our model allows us to express instantiation and deployment semantics that cannot easily be expressed in AspectJ.

When using AspectJ's `perThis` or `perTarget` modifiers, state can be only associated with objects that are caller or callee, respectively, in a pointcut. In CAESAR, state can be associated with arbitrary objects and arbitrary relations between objects. For example, we could associate state with every *pair* of `this` and `target`, or with any argument of a method call. In the `percflow` case we can either simulate the AspectJ semantics but we could also do something more sophisticated, e.g., deploy an instance of an optimization aspect only if the number of calls to the method to be optimized is executed more than a certain threshold.

5.5 Related Work

Open classes: An open class is a class to which new fields or methods can be added without editing the class directly. For example, in MultiJava [CLCM00] additional methods can be attached to a class. In AspectJ, methods as well as fields can be added to a class by means of *introductions*. As already discussed in Sec. 5.2, open classes are in contrast to the concept of independent extensibility [Szy96], an essential prerequisite for reusable and extensible software. We think that CAESAR combines both the advantages of open classes *and* independent extensibility.

Adaptive Plug and Play Components (APPCs) [ML98] and their aspect-oriented variant of *Aspectual Components* [LLM99] are related to our work in that both approaches support the definition of multi-abstraction components / aspects and have a vague definition of required and provided interfaces. However, the latter feature was not well integrated with the type system. Recognizing this deficiency, the successor model of *Pluggable Composite Adapters (PCAs)* [MSL01] even dropped this notion and reduced the declaration of the expected interface to a set of standard abstract methods. With the notion of collaboration interfaces, CAESAR represents a qualitative improvement over all three models, as far as support for multi-abstraction aspects is concerned.

Due to the lack of a notion similar to collaboration interfaces, component implementations and their bindings are coupled. Hence, connectors and adapters in APPC, Aspectual Components, and PCA models are bound to a fixed implementation of an aspect and cannot be reused. In addition,

these approaches rely on a dedicated mapping sublanguage that is less powerful than our notion of object-oriented wrappers with wrapper recycling. Among these approaches, the APPC and Aspectual Component model have no notion of aspect instantiation and deployment. Base and aspect code are statically merged at compile-time, which corresponds to the open class style of AspectJ. As already discussed in Sec. 5.2 and in the first paragraph of this section, open classes do not properly support independent extensibility [Szy96].

There are two more consequences of the open-class-like approach taken by APPCs and Aspectual Components: (a) it is not possible to instantiate several instances of the same aspect type for the same set of basis objects, and (b) aspectual polymorphism is not supported. From the three approaches discussed in this and the previous paragraph, only PCAs share with CAESAR the notion of explicit instantiation of aspects and some kind of deployment concept. However, only a restricted form of dynamic deployment is supported. Furthermore, the PCA model, as well as APPCs, have no support for pointcuts and advices. Finally, the lack of the notion of virtual types is another drawback of these approaches as compared to the work presented here.

Lasagne [TVJ⁺01] is a runtime architecture that features aspect-oriented concepts. An aspect is implemented as a layer of wrappers. Aspects can be composed at run-time, enabling dynamic customization of systems, and context-sensitive selection of aspects is realized, enabling client-specific customization of systems. Although Lasagne is an architectural approach focusing on middleware (instead of a general purpose language extension as CAESAR), it has some similarity with CAESAR. In particular, Lasagne also features extensions that are created and deployed at runtime, and it also provides means to restrict the visibility of an extension to a particular scope (as our `deploy` block statement).

In [BA01] an extension of the composition filter model [ABV92] geared more towards aspect-oriented programming is discussed. With composition filters, it is possible to define various filters for incoming and outgoing messages of an object. By means of *superimposition* [BA01], it is possible to apply these filters to objects that are specified via a join point declaration similar to AspectJ pointcuts. Composition filters have no dedicated means to separate aspect implementation and binding, and there is notion of deployment or aspectual polymorphism. In comparison with CAESAR, where almost everything is specified as usual OO code, composition filters are more declarative. On one hand, this makes it easier to express kinds of concerns that are easily expressible with the declarative sublanguage, but on the other hand it restricts applicability to arbitrary kinds of concerns.

5.6 Chapter Summary

In this chapter, we argued that join point interception (JPI), that is, intercepting and eventually modifying the execution of running code at certain points, alone does not suffice for a modular structuring of aspects, resulting in tangled aspect code. We discussed several problems resulting from the lack of an appropriate higher-level module construct on top of join points and advices. Our solution is based on CAESAR as introduced in the previous chapter. With the additional language means introduced in this chapter, CAESAR is a model for aspect-oriented programming with a higher-level module concept on top of JPI, which enables reuse and componentization of aspects, allows us to use aspects polymorphically, and introduces a novel concept for dynamic aspect deployment.

*The end always passes
judgement on what has gone
before.*

Publilius Syrus (~100 BC)

CHAPTER 6

Conclusions

In Chap. 1, this thesis has been motivated from a purely *conceptual* point of view: It should be easier to reason about models of reality in terms of programming language constructs. In order to do this, we need constructs that enable a smooth transition from conceptual hierarchies to module hierarchies. This thesis is concluded by a more abstract reflection on the different *technical* ideas underlying the models proposed in Chap. 2–5.

Indeed, there are some ‘red threads’, some recurring problems and ideas, that pervade the models proposed in Chap. 2–5. The identification of these red threads has the following purposes:

- The proposed models can be viewed from a new technical, but nevertheless abstract, perspective, thereby yielding new insights and possibly generating new ideas.
- Identify important areas of future research in programming language design.

In the following, each of these key ideas will be presented, together with a discussion why these key ideas are of pivotal importance in the context of this thesis and why they may be of interest for future programming language research.

6.1 Transparent Redirection

The term ‘transparent redirection’ has already been used in Chap. 2 to denote a particular property of our refined inheritance mechanism, namely to let `B`’s `this` refer to `M(B)` if the two modules `M` and `B` are combined. In

this section, we generalize this term to mean any mechanism that improves the decoupling of a name from its denotation.

In object-oriented programming, this idea manifests itself as late binding of method calls. In Chap. 2, we isolated the redirection property of late binding from its other implications. With delegation layers and its incorporation of family polymorphism, the idea of transparent redirection is also applied to types - modules that use type variables are not aware of the concrete type that will be used at runtime. Similarly, aspect bindings and implementations in CAESAR are not aware of the concrete realizations of their counterpart.

Due to the incorporation of join points and advice, CAESAR also offers yet another interesting interpretation of transparent redirection, which is similar to traditional late binding – the difference between these two forms of transparent redirection is in the *scope* of the redirection.

With traditional late binding, only method calls to objects that have explicitly been created as instances of a subclass *S* of a class *C* are redirected to the definitions in *S*, whereas with advices, all advised method calls that are in the scope of a deployed aspect are redirected to their extended definitions in the aspect, regardless of the class that was used to create these objects. On a syntactic level, the similarity becomes most striking if one considers embedded advices as introduced in Fig. 5.9 on page 143.

Transparent redirection can also be regarded as a form of polymorphism – the same name may denote different things if it is evaluated in different places or at different points in time. The pivotal importance of transparent redirection to aspect-oriented programming has also been documented by Filman and Friedman [FF00, Fil01], who identify what they call “*obliviousness*” or “*implicit invocation*” as a defining property of aspect-oriented programming.

6.2 Incremental Specification, Increment Combination, and Subsumption

The ability to specify a module *B* in terms of the difference $\Delta(B, A)$ to another module *A*, *incremental specification*, is one of the most important ideas in programming languages. Its main benefits are:

- **Reuse:** We do not have to repeat specifications of *B* that are already defined in *A*.
- **Sharing:** Changes and improvements to *A* are automatically reflected in *B*.

A related, though less known idea is that of *increment combination*, that is, the ability to combine the increments in modules *B1* and *B2* in a module *C*,

such that \mathbf{C} corresponds to $\mathbf{A} + \Delta_1(\mathbf{B}, \mathbf{A}) + \Delta_2(\mathbf{B}, \mathbf{A})$, whereby the meaning of $+$ depends on the specific combination mechanism at hand.

Incremental specification can also be seen as a tool to conveniently specify *variants* of a module, e.g., module \mathbf{B} as above adds or refines features of module \mathbf{A} . Increment combination can hence be seen as variant combination, that is, the additions and refinements of the variants are combined.

Although incremental specification and increment combination are already useful in their own, their real power unfolds if these mechanisms are combined with *subsumption*. Subsumption comes in two different flavors: *internal subsumption* and *external subsumption*. External subsumption means that a variant of a module can be used in places where the original module is expected. Internal subsumption means that subsumption is also available when combining different increments in the sense that the code that combines the increments is not necessarily aware of the exact shape of the increments it combines and can thus be used polymorphically with respect to the increments to be combined.

In all parts of this thesis the principles of incremental specification, increment combination, and subsumption played a leading role. The best-known mechanism for incremental specification is inheritance in object-oriented languages. Sample mechanisms for increment combination are multiple inheritance and mixin-inheritance. External subsumption is usually realized in the form of subtype polymorphism. Internal subsumption is not available in any of these mechanisms because all modules that are composed are known statically.

In Chap. 2, we refined the usual object-oriented inheritance mechanism in two ways. First, by giving the programmer fine-grained control over the semantics of the desired composition, hence widening the range of mechanisms for expressing the Δ . Second, by combining inheritance with object composition, thereby enabling increment composition, as well, through the presence of subtype polymorphism. Since both the individual objects that are combined and the compound objects are only known by upper bound (subtype polymorphism), both internal and external subsumption are supported.

One of the problems with inheritance is that only single classes can be specified incrementally. With delegation layers (Chap. 3), a generalized notion of inheritance has been proposed that allows to specify a *set* of classes incrementally as the Δ to another set of classes. Due to an appropriate generalization of subtype polymorphism to such sets of classes, both increment combination (due to polymorphic assignment of ‘parent’ layers) and subsumption (both internal and external due to subtype polymorphism) is supported.

In CAESAR, incremental specification is supported on various levels (cf. Sec. 4.4): collaboration interfaces, bindings, and implementations can all be specified relative to an ‘inherited’ CI, binding, and implementation, re-

spectively. Due to appropriate subtyping relations on bindings and implementations, external subsumption is supported on all these levels, too. The composition of a binding and a CI implementation can be seen as a form of static increment composition. In Sec. 4.5, we indicated how dynamic increment combination and internal subsumption can be supported as well.

With respect to the introduction of pointcuts as in Chap. 5, an interesting topic for future research will be how joinpoints can be better structured by means of incremental specification. Currently, incremental specification as well as increment composition is supported by means of logical (*and*, *or*, *not*) pointcut composition operators. However, in the authors opinion, this is just the beginning of better structuring mechanisms for the specification of join points. For example, it is currently not possible to reason about sets of related pointcuts in an incremental way: If **p1**, **p2**, and **p3** are three related pointcuts such that **p2** and **p3** are specified incrementally in terms of **p1**, it is not possible to refine the whole ‘hierarchy’ of pointcuts by just ‘overriding’ the definition of **p1**: There is no ‘late binding’ of **p1** inside the definitions of **p2** and **p3**¹.

6.3 More Powerful Interfaces

The idea of separating the interface and the implementation of a module in order to foster information hiding dates back to the 70’s [Par72b] and was pioneered by languages such as Modula [Wir82] and ADA [DoD83]. In these early days of modules an interface to a module was basically a list of functions or procedures. Object-oriented languages improved this concept by the concept of a class that can be instantiated multiple times and may be related to another class via subtyping.

Indeed, subtyping is the primary tool for information hiding in object-oriented languages. For example, Java has a dedicated language notion of object-oriented interfaces, such that any class can declare itself as a subtype of these interfaces. In C++, fully abstract classes are used for the same purpose. However, the usual notion of interfaces has proved to be not powerful enough in several places in the course of this thesis. Their shortcomings can be summarized in two points:

- **Lacking support for “Hollywood-style” communication:** The degree to which a module is reusable depends on the ability of the module to be parameterized with respect to the parts of the module that are specific to the context of usage. The simplest form of parameterization is procedural/functional/lambda abstraction, whereby the parameters are the respective arguments. This is also the only kind of

¹AspectJ has the notion of *abstract pointcuts* that can be used for ‘late binding’ of pointcuts. However, this would require **p1** to be empty.

parameterization that is available if one wants to communicate with a module via an interface.

The ability to parameterize a module by overriding parts of its definition, as exemplified by method overriding in OO languages, is one of the most important improvements over pure functional abstraction. This can also be called Hollywood-style communication (‘don’t call us, we call you’) because client-specific definitions are called during the evaluation of a module-specific definition. However, conventional interfaces offer no support whatsoever to support such Hollywood-style communication. This proved to be a problem in several places. All delegation-based approaches suffer from the problem that conventional interfaces cannot describe requirements, e.g., abstract methods, of a parent to a child object. In Sec. 2.6 an explicit notion of a *specialization interface* with its own subtyping relation, that may be different from the usual subtype relation, has been proposed. In the context of CAESAR, dedication notions of *provided* and *expected* methods in interfaces have been proposed in order to foster better bidirectional, Hollywood-style communication.

- **Lacking support to describe multi-abstraction modules:** If a module consists of multiple collaborating abstractions, these abstractions can be described by a corresponding set of interfaces, but if there is no representation of their togetherness, it is hard to reason about the group of interfaces as a whole, e.g., in order to use a multi-abstraction module polymorphically. Since both delegation layers and CAESAR are about multi-abstraction modules, more sophisticated mechanisms were necessary. The primary idea used in these approaches to describe the interface of a multi-abstraction module was to introduce types as first class properties of other, higher-order types, technically realized by means of virtual classes, dependent types, and family polymorphism. However, dependent types and family polymorphism are a very new concept and their possibilities, enhancements, limitations and semantic implications are just beginning to be explored [OCRZ03].

6.4 Runtime Composition and Static Typing

The last important recurring theme in this thesis has been the tension between composing modules at runtime on one hand and static typing on the other hand. This tension is due to the fact that, from a computational point of view, it is desirable to defer the composition of modules until *runtime*, thereby enabling to choose the ‘ingredients’ of the composition *dynamically*. However, from a typing point of view, we want to know as much properties about the composed modules at *compile time* in order to maximize the

number of type-safe programs that can be proved type-safe *statically*.

Indeed, retaining static type safety has been one of the hardest issues recurring in this work. In Sec. 2.5 we proposed a mechanism to reconcile dynamic specialization and static typing. In Sec. 2.6 we demonstrated that abstract classes and method header specializations need special consideration in the presence of dynamic specialization, too. With delegation layers and CAESAR, the introduction of dependent types and family polymorphism was necessary in order to use sets of classes polymorphically in a statically safe way.

6.5 Summary and Future Work

This goal of this thesis has been to improve the state-of-the-art with respect to both hierarchical and crosscutting decomposition mechanisms. Based on the most successful paradigm for separation of concerns, object-oriented programming, a three chord of proposals that refine, generalize and extend the object-oriented composition concepts with respect to both hierarchical and crosscutting decomposition have been proposed.

With compound references a new abstraction for object references that unifies aggregation, inheritance and delegation has been presented. It provides explicit linguistic means for expressing and combining individual composition properties on-demand, thereby enriching the spectrum of composition semantics that can be expressed seamlessly in the programming language.

In the delegation layer approach, it has been explored how the means for hierarchical decomposition can be generalized to work on sets of collaborating classes, motivated by the observation that a slice of behaviour affecting a set of collaborating classes is a better unit of organization and reuse than single classes. Delegation layers scale the OO mechanisms for single objects, such as delegation, late binding, and subtype polymorphism, to sets of collaborating objects.

With *Caesar*, ideas for hierarchical decomposition from the first two parts of the thesis have been extended and applied in order to provide means for composing crosscutting hierarchies. Caesar's strengths are in the reuse and componentization of aspects, allowing to use aspects polymorphically. The notion of aspectual polymorphism as a generalization of subtype polymorphism to crosscutting models has been introduced and a novel concept for dynamic aspect deployment has been proposed.

Some concrete areas of future research have already been indicated in the course of this chapter. The language means to specify pointcuts can be improved in various ways. Means to specify pointcuts incrementally and to combine these increments are desirable. Once a proper notion of late binding for pointcuts has been explored, the well-known concepts for incremental

specification and increment combination known for late-bound methods can potentially be transferred to pointcut definitions. For example, if pointcuts could be overridden in subclasses, such that the definitions of all other pointcuts and advices that refer to the overridden pointcut are late bound to the current definition, different increments could be combined similar to method combination with multiple inheritance or mixin inheritance. Another interesting area of future research is the design of more abstract, *intentional pointcuts*, that are less coupled to the syntax tree of a program. The ability to specify pointcuts that refer to individual runtime entities such as objects will also be an important issue that brings us back to the ‘runtime versus static typing’ conflict outlined in the previous section.

A more formal treatment of the ideas presented in this thesis will be beneficial to understand their respective implications in more detail. All language extensions in the different chapters have been presented as extensions to the Java programming language. This is useful for concrete implementations and for their evaluation in practice. However, the usage of a very simple formal language like Featherweight Java [IPW99] will be helpful to demonstrate the core ideas more concisely and precisely.

Bibliography

- [AB92] M. Aksit and L. Bergmans. Obstacles in object-oriented software development. In *Proceedings OOPSLA '92. ACM SIG-PLAN Notices 27(10)*, pages 341–358. ACM, 1992.
- [ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings ECOOP '92. LNCS 615*, pages 372–395. Springer, 1992.
- [AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
- [ALZ00] D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In *Proceedings ECOOP 2000. LNCS 1850*, pages 154–178. Springer, 2000.
- [Asp03] AspectJ homepage, 2003. <http://aspectj.org>.
- [AWB⁺93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*. Springer, 1993.
- [BA01] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters, 2001. Available at trese.cs.utwente.nl/composition_filters/.
- [BC89] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA '89. ACM SIG-PLAN Notices 24(10)*, pages 1–6, 1989.

- [BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90. ACM SIGPLAN Notices 25(10)*, pages 303–311. ACM, 1990.
- [BCML02] A. Brown, R. Cardone, S. McDirmid, and C. Lin. Using mixins to build flexible widgets. In *Proceedings AOSD '02*, pages 76–85. ACM, 2002.
- [Ber90] L. M. Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *Proceedings of ECOOP/OOPSLA '90, ACM SIGPLAN Notices 25(10)*, pages 181–193, 1990.
- [BFK00] K. Beck, M. Fowler, and J. Kohnke. *Planning Extreme Programming*. Addison-Wesley, 2000.
- [BH02] J. Baker and W. C. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings AOSD '02*, pages 86–95. ACM, 2002.
- [BI94] K. Baclawski and B. Indurkha. The notion of inheritance in object-oriented programming. *Communications of the ACM*, 37(9):118–119, 1994.
- [BL92] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, 1992.
- [BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Macmillan, 1976.
- [BMR⁺96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. Wiley, 1996.
- [Bos98] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.
- [BOW98] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98. LNCS 1445*, pages 523–549. Springer, 1998.
- [Box97] D. Box. *Essential COM*. Addison-Wesley, 1997.
- [Bra83] R. J. Brachman. What IS-A is and isn't: An analysis of taxonomic link in semantic networks. *Computer*, 16(10):30–36, 1983.
- [Bro75] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.
- [Bro87] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

BIBLIOGRAPHY

- [BW00] M. Büchi and W. Weck. Generic wrappers. In *Proceedings ECOOP '00. LNCS 1850*, pages 201–225. Springer, 2000.
- [CCH⁺89] P. S. Canning, W. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FCPA) '89*, pages 273–280, 1989.
- [CG90] B. Carré and J. Geib. The point of view notion for multiple inheritance. In *Proceedings OOPSLA/ECOOP '90. ACM SIGPLAN Notices 25(10)*, pages 312–321. ACM, 1990.
- [CH88] R. Chaffin and D. J. Herrmann. The nature of semantic relations: A comparison of two approaches. In *Relational Models of the Lexicon: Representing Knowledge in semantic networks*, pages 289–334. Cambridge University Press, 1988.
- [Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings ECOOP '92, LNCS 615*, pages 33–56. Springer, 1992.
- [Cha93] C. Chambers. Predicate classes. In *Proceedings ECOOP '93, LNCS 707*, pages 268–297. Springer, 1993.
- [CHP99] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proceedings Conference on Programming Language Design and Implementation (PLDI) '99*, pages 50–63. ACM, 1999.
- [CL01] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proceedings of International Conference on Software Engineering (ICSE) '01*, pages 285–294. IEEE Computer Society, 2001.
- [CLCM00] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings OOSPLA '00, ACM SIGPLAN Notices 35(10)*, pages 130–145, 2000.
- [DDH72] O.-J. Dahl, E. W. Dijkstra, and C. Hoare. *Structured Programming*. Academic Press, 1972.
- [Dij68] E. W. Dijkstra. The structure of the “THE”-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.
- [Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [DoD83] US Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD1815 A.

- [Ern99] E. Ernst. *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.
- [Ern01] E. Ernst. Family polymorphism. In *Proceedings ECOOP '01*, LNCS 2072, pages 303–326. Springer, 2001.
- [ES95] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1995.
- [FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns at OOPSLA '00*, 2000.
- [Fil01] R. E. Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns at ECOOP '01*, 2001.
- [FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL) '98*, pages 171–183. ACM, 1998.
- [Fow99] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [Gam98] E. Gamma. Extension object. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design*, pages 79–88. Addison-Wesley, 1998.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
- [GKN92] D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *Computer*, 25(6):30–38, 1992.
- [Hau93] F. J. Hauck. Inheritance modeled with explicit bindings: An approach to typed inheritance. In *Proceedings OOPSLA '93. ACM SIGPLAN Notices 28(10)*, pages 231–239. ACM, 1993.
- [HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90. ACM SIGPLAN Notices 25(10)*, pages 169–180. ACM, 1990.
- [HK02] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA '02. ACM SIGPLAN Notices 37(11)*, pages 161–173. ACM, 2002.

BIBLIOGRAPHY

- [Höl93] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings ECOOP '93*, LNCS 707, pages 36–56. Springer, 1993.
- [HM00] S. Herrmann and M. Mezini. PIROL: A case study for multi-dimensional separation of concerns in software engineering environments. In *Proceedings of OOPSLA 2000. ACM SIGPLAN Notices 35(10)*, pages 188–207. ACM, 2000.
- [HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93. ACM SIGPLAN Notices 28(10)*, pages 411–428, 1993.
- [Hol92] I. M. Holland. Specifying reusable components using contracts. In *Proceedings ECOOP '93. LNCS 615*, pages 287–308, 1992.
- [HOT97] W. Harrison, H. Ossher, and P. Tarr. Using delegation for software and subject composition. Technical Report RC 20946(92722), IBM Research Division T.J. Watson Research Center, Aug 1997.
- [IPW99] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 1999.
- [JFC] Java Foundation Classes. <http://java.sun.com/products/jfc/>.
- [Kic01] G. Kiczales. Aspect-oriented programming - the fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, Vanderbilt University, Nashville, Tennessee, December 13-14, 2001.
- [KLM⁺97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP'97*, LNCS 1241, pages 220–242. Springer, 1997.
- [Kni99] G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings ECOOP '99*, LNCS 1628, pages 351–366. Springer, 1999.
- [Kni00] G. Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, University of Bonn, Institute for Computer Science III, 2000.
- [Lak90] G. Lakoff. *Women, Fire, and Dangerous Things - What categories reveal about the mind*. University of Chicago Press, 1990.

- [Lar01] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2001.
- [Lie86] H. Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings OOPSLA '86. ACM SIGPLAN Notices 21(11)*, pages 214–223, 1986.
- [LKA⁺95] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, March 1999.
- [LLP⁺88] R. Lampert, D. Littman, J. Pinto, E. Soloway, and S. Letovsky. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), 1988.
- [Läm02] R. Lämmel. A semantical approach to method-call interception. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '02*, pages 41–55. ACM, 2002.
- [Mar96] R. C. Martin. Granularity. *C++ Report*, 8(11), 1996. www.objectmentor.com/publications/granularity.pdf.
- [MBF99] M. Mattson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):80–87, October 1999.
- [McI68] M. McIlroy. Mass produced software components. In P. Naur and B. Randell, Software Engineering - Report on a conference sponsored by the NATO Science Committee, 1968. Garmisch, Germany.
- [Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
- [Mez97] M. Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97. LNCS 1241*, pages 190–219. Springer, 1997.
- [Mez98] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publisher, 1998.
- [MFH01] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings of OOPSLA '01, ACM SIGPLAN Notices 36(11)*, pages 211–222, 2001.

BIBLIOGRAPHY

- [MK96] J. Magee and J. Kramer. Dynamic structure in software architecture. In *Proceedings of the ACM SIGSOFT '96 Symposium on Foundations of Software Engineering (FSE)*, 1996.
- [MK03] H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In *To be published in Proceedings ECOOP '03*, Springer LNCS, 2003.
- [ML98] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98. ACM SIGPLAN Notices 33(10)*, pages 97–116, 1998.
- [MMP89] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89. ACM SIGPLAN Notices 24(10)*, pages 397–406, 1989.
- [MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.
- [MO02] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02, ACM SIGPLAN Notices 37(11)*, pages 52–67, 2002.
- [MO03] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.
- [MOT97] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *Proceedings of International Conference on Software Engineering (ICSE) '97*, pages 692–700. IEEE Computer Society, 1997.
- [MPG⁺00] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings ECOOP '00, LNCS 1850*, pages 337–361. Springer, 2000.
- [MSL01] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001.
- [OCRZ03] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *To appear in Proceedings ECOOP '03*. Springer LNCS, 2003.

- [OM01] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings OOPSLA '01*, ACM SIGPLAN Notices 36(11), pages 283–299, 2001.
- [OMG99] Object Management Group. *CORBA Components Final Submission*, 1999.
- [Ost02] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP '02. LNCS 2374*, pages 89–110. Springer, 2002.
- [OT00] H. Ossher and P. Tarr. On the need for on-demand remodularization. In *ECOOP'2000 workshop on Aspects and Separation of Concerns*, 2000.
- [Par72a] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [Par72b] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.
- [Ped89] C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices 24(10)*, pages 407–417, 1989.
- [PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings AOSD '02*, pages 141–147. ACM, 2002.
- [PSDF01] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings REFLECTION '01, LNCS 2192*, pages 1–24, 2001.
- [Ree95] T. Reenskaug. *Working with Objects: The OOram software Engineering Method*. Manning, 1995.
- [RG98] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices 33(10)*, pages 117–133, 1998.
- [Ros28] W. D. Ross. *The Works of Aristotle, Volume 1: Logic*. Oxford University Press, 1928.
- [Sak89] M. Sakkinen. Disciplined inheritance. In *Proceedings ECOOP '89*, pages 39–56. Cambridge University Press, 1989.

BIBLIOGRAPHY

- [SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98, LNCS 1445*, pages 550–570, 1998.
- [SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. PrenticeHall, 1996.
- [SL86] E. Soloway and S. Letovsky. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.
- [SM95] P. Steyaert and W. D. Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95. LNCS 952*, pages 127–144. Springer, 1995.
- [SPL98] L. M. Seiter, J. Palsberg, and K. Lieberherr. Evolution of object behavior using context relations. *IEEE Transactions on Software Engineering*, 24:79–92, 1998.
- [SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Vol. 2*. Wiley, 2000.
- [Sto93] V. C. Storey. Understanding semantic relationships. *Very Large Databases Journal*, 2(4):455–488, 1993.
- [Sun] Sun Microsystems. Java 2 SDK Documentation. <http://java.sun.com/j2se/1.4/docs/index.html>.
- [Syn87] A. Synder. Panel on inheritance. In *Addendum to the Proceedings of OOPSLA '87. ACM SIGPLAN Notices 23(5)*, 1987.
- [Szy96] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings 19th Australian Computer Science Conference*. Australian Computer Science Communications, 1996.
- [Szy98] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [Tai96] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):439–479, 1996.
- [Tho97] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97. LNCS 1241*, pages 444–471, 1997.
- [TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings International Conference on Software Engineering (ICSE) '99*, pages 107–119. ACM Press, 1999.

- [Tor98] M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, 1998.
- [TVJ⁺01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Joergensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the International Conference on Software Engineering (ICSE) '01*, pages 233–242. IEEE Computer Society, 2001.
- [US87] D. Ungar and R. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87. ACM SIGPLAN Notices 22(12)*, pages 227–242, 1987.
- [VN96] M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices 31(10)*, pages 359–369, 1996.
- [Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1:7–87, August 1990.
- [Wel95] C. Welty. *An Integrated Representation for Software Development and Discovery*. PhD thesis, RPI Computer Science Dept, July 1995.
- [Wik] Wikipedia, the free encyclopedia. www.wikipedia.org.
- [Win92] J. Winkler. Objectivism: “class” considered harmful. *Communications of the ACM*, 35(8):128–130, 1992.
- [Wir71] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), 1971.
- [Wir82] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1982.
- [Wol97] D. Wolber. Reviving functional decomposition in object-oriented design. *Journal of Object-Oriented Programming*, 10(6):31–38, 1997.

Lebenslauf

Personalien

Name: Klaus Ostermann
Anschrift: Barkhausstr. 22
64289 Darmstadt
Telefon: 06151 - 6696637
Geburtstag: 6. Dezember 1974
Geburtsort: Cloppenburg
Staatsangehörigkeit: deutsch
Familienstand: ledig

Werdegang

1981-1987: Grundschule und Orientierungsstufe in Lindern
1987-1994: Gymnasium in Löningen, Abschluß Abitur
1994-1995: Wehrdienst
1995-2000: Studium der Informatik an der Universität Bonn
02.01.2001: Diplom Informatik
2001-2002: Doktorand bei Siemens AG, München
seit 01.01.2003: Wissenschaftl. Mitarbeiter am Fachgebiet
Softwaretechnologie, Fachgebiet Informatik, TU-Darmstadt